Efficient Utilization of GPGPU Cache Hierarchy

Mahmoud Khairy
Computer Engineering
Cairo University, Egypt
makhairy@eng.cu.edu.eg

Mohamed Zahran
Computer Science
New York University, U.S.
mzahran@cs.nyu.edu

Amr G. Wassal
Computer Engineering
Cairo University, Egypt
wassal@eng.cu.edu.eg

ABSTRACT

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

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

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

Categories and Subject Descriptors

C.1.4 [**Computer System Organization**]: Processor Architecture—*Parallel Architecture*; B.3.2 [**Memory Structures**]: Design Style—*Cache memories*

GPGPU-8, February 7, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3407-5/15/02...$15.00
http://dx.doi.org/10.1145/2716282.2716291

General Terms

Design, Performance

Keywords

Cache Management, GPGPU, Warp Throttling, Conflict-avoiding, Cache Bypassing

1. INTRODUCTION

Throughput-oriented processors, such as General Purpose Graphics processing Units (GPGPUs), have been widely adopted for accelerating compute-intensive data-parallel applications due to their high computational power and energy efficiency [5, 35, 4, 21, 8, 30]. However, GPGPU programming is a difficult task. The programmer has to explicitly manage the on-chip scratchpad memory to generate coalesced memory accesses and exploit data locality[23, 36]. Further, it has been shown that the memory throughput has become a limiting factor for many GPGPU applications performance [21]. To address these issues, modern GPUs [44, 33] are equipped with general purpose on-chip cache hierarchy in an attempt to reduce off-chip memory bandwidth demand, increase memory system throughput, improve the performance of some irregular GPGPU applications and enhance the GPU programmability.

GPU cache size is very limited, compared to the number of active threads GPU executes concurrently. For instance, NVIDIA's Fermi GPU [44] supports 1536 active threads per core, and L1 cache size is configurable to 16KB or 48KB. Thus, the average L1 cache capacity per thread is only 10 or 32 bytes, which is less than a single cache line size (=128 bytes). This behavior is also found in NVIDIA's Kepler GPU [33] that has 2048 active threads per core and a read-only L1 data cache of size 48KB. This means that the GPU cache is not designed to keep the per-thread working set, as it is the case in CPU (for example, Intel core i7 CPU [14] contains 2 threads per core, 32KB L1, thus 16KB per thread). In fact, GPU caches were designed to exploit some access patterns that exhibit small cache footprint per thread and can fit in the cache (e.g. spilled registers, small-stride access pattern [36] and inter-core data locality [33]). In case GPGPU applications rely on caches to exploit data locality and they contain a large cache footprint per-thread, the active threads will compete on the few available cache lines and the L1 cache will be susceptible to *thrashing*. Moreover, the limited number of set associativity, typically between 4-6 [29], makes the L1 cache more vulnerable to *associativity* stalls and *conflict* misses. In addition, many GPGPU

applications use the scratchpad memory to exploit locality. These applications show a *streaming* behavior when they are cached. Cache management schemes that are unaware of these streaming applications causes useless unintended contention at L1 cache and this may hurt the performance instead.

CTA throttling [20, 26], Warp throttling [39, 40, 27, 45], FIFO buffer [18] and thrashing-resistant cache replacement policy [6, 27, 7] are different techniques have been proposed to alleviate cache thrashing, while cache bypassing [45, 27, 18] was proposed to mitigate the associativity stalls. However, many of these proposals address the cache thrashing problem only, incur a considerable storage overhead and require significant changes to the baseline architecture. Further, the proposed cache bypassing to handle associativity stalls is not an effective method to efficiently utilize the available cache resources. In many cases, bypassing occurs while cache sets are underutilized.

In this work, we propose a low-cost thrashing-resistant conflict-avoiding streaming-aware GPGPU cache management scheme that efficiently utilize the GPGPU cache resources. The proposed method employs three techniques. First, it dynamically detects and bypasses streaming applications that show streaming behavior in L1 or L2 cache. Second, we propose dynamic warp throttling via cores sampling (DWT-CS) to alleviate cache thrashing. DWT-CS monitors the MPKI at L1, when it exceeds a specific threshold, all GPU cores are sampled with different number of active warps. Then, the active warps per all cores will be throttled to the number of warps that is associated with the winner core (the core achieved the highest performance during the sampling period). Third, we employ a better cache indexing function, Pseudo Random Interleaving Cache (PRIC), that is based on polynomial modulus mapping [38], to mitigate associatively stalls and eliminate conflict misses. PRIC near-randomly and fairly distributes memory accesses over cache sets and thus efficiently utilizing the cache resources.

This paper makes the following contributions:

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

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

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

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

The rest of this paper is organized as follows, Section 2 describes our baseline architecture and methodology, Section 3 describes workload characterization, Section 4 describes our proposed techniques, Section 5 presents the experimental results, Related works are discussed in Section 6, and Section 7 concludes.

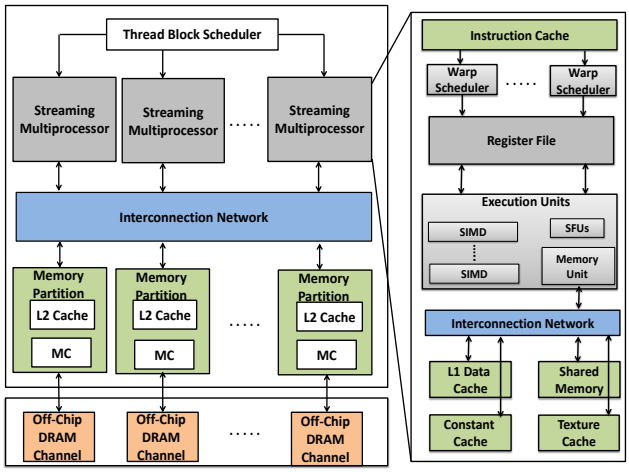


Figure 1: Baseline GPGPU Architecture

2. BACKGROUND AND BASELINE ARCHITECTURE

2.1 GPGPU Programming Model

The CUDA [31] or OpenCL [34] programming model allows the programmers to express the data level parallelism in terms of fine-grain scalar threads. A typical GPGPU application consists of multiple kernels (or grids). Each kernel contains a group of thread blocks or cooperative thread array (CTA). Each thread block is composed of 3-dimensional scalar threads. Threads within the same thread block communicate with each other through a shared on-chip scratchpad memory and synchronization primitives. During runtime, each consecutive 32 threads are grouped together to formulate a warp (a.k.a. wavefront). Warps are executed in a single instruction multiple-threads (SIMT) model. In SIMT execution model: all threads within the same warp execute the same PC (i.e. execute in a lock-step), threads are allowed to follow different control flow paths and a long memory latency is tolerated by a zero-overhead warp context switching.

2.2 Baseline Architecture

Our baseline GPGPU, shown in Figure 1, consists of multiple GPU cores, named Streaming Multiprocessors (SMs), [1]and a group of memory partitions. Each SM has its own private L1 data cache, read-only texture cache, constant cache and software-managed scratchpad memory, named shared memory. They also contain a group of execution units, such as single instruction multiple data units (SIMDs) and special function units (SFUs). Each memory partition has a slice of the L2 cache and a GDDR5 memory controller that are shared among the SMs. The SMs and the memory partitions are connected via an on-chip network.

A thread block scheduler, as shown in Figure 1, distributes the thread blocks among SMs in a load-balanced round-robin [2] fashion. Thread block is dispatched to a SM only if the required resources of the thread block are available on this SM

[1]In this paper, we use the terms GPU core and Streaming Multiprocessors interchangeably.

Table 1: Simulated baseline GPGPU configuration

SM configuration	15 SMs, 700 MHZ, 1536 threads, 32 threads/warp, 48 warp/SM, SIMD width = 32, 5-Stage Pipeline, 32684 registers
L1 Cache	16KB/4-way/128B/global-write-evict-local-write-back/no-write-allocate/allocate-on-miss/32 MSHR entries
L2 Cache	6 partitions x 128KB/16-way/128B line/write-back/write-allocate/ 32 MSHR entries
Shared Memory	48 KB
Constant Cache	8KB
Texture Cache	12KB/24-way/128B line
# Warp scheduler	2 per SM (24 warps per scheduler)
Warp scheduling	Greedy-then-oldest (GTO) [39]
Branch Divergence	PDOM [11]
Interconnect	1 crossbar/direction, 32B channel width, 1400 MHZ
Memory Model	6 GDDR5 Memory Controllers (MCs), First-Ready FCFS (FR-FCFS) scheduling, 924 MHz, BW=179.2 GB/s
GDDR5 Timing	tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6

(e.g. register file, shared memory, warp scheduler entries, etc.). Thread block are subdivided by hardware into warps. Each SM contains a number of warp scheduler. The warp scheduler employs a greedy-then-oldest (GTO) scheduling policy. GTO runs a single warp until it stalls then picks the oldest ready warp [39]. Our baseline handles control flow divergence and re-convergence with a post dominator (PDOM) re-convergence stack [11]. Each SM contains a memory-coalescing unit that attempts to coalesce memory requests of active threads within each warp into the fewest possible cache line-sized memory requests.

2.3 Methodology

We simulate the baseline architecture using GPGPU-Sim v3.2.1 [2], a publicly-available cycle-accurate GPGPU simulator. The GPU simulator is configured to be similar to NVIDIA Fermi GTX480 [44]. We use the configuration file provided with GPGPU-Sim without any modifications. The configuration parameters are described in Table 1. The simulator was modified to implement the proposed techniques that we evaluate in this work.

We considered a wide range of GPGPU CUDA workloads, including applications from Rodinia [3], Poly-Bench [13] and NVIDIA SDK [32]. NN, IIX, SPMV_S, and KM are adopted from GPGPU-sim workloads [2], MapReduce [16], SHOC [9], and CCWS applications suite [39] respectively. In total, we study 22 applications described in Table 2. The applications run until completion, with the exception of SYRK, GESUMMV, and SCLUSTER, due to the long simulation time of these applications, we execute SYRK and GESUMMV only up to 100 million instructions, while SCLUSTER up to 300 million instructions.

Table 2: GPGPU Workloads

Name	Abbrev.	Type
Black Scholes [32]	BLK	Streaming
Scalar Product [32]	Sprod	Streaming
Vector Addition [32]	VAdd	Streaming
Fast Walsh Transform[32]	FWT	Streaming
Needleman-Wunsch [3]	NW	Streaming
Hot Spot [3]	HS	Streaming
Separable Convolution [32]	CONV	Streaming
Structured grid [3]	SRAD	Conflict
3D Stencil [3]	3DS	Conflict
2D Convolution [13]	2DCONV	Conflict
2 Matrix Multiplication [13]	MM	Conflict
Stream Cluster [3]	SCLUSTER	Conflict
Breadth First Search [3]	BFS	Thrashing
Sparse Matrix Vec. Mult.[9]	SpMV	Thrashing
Inverted Index [16]	IIX	Thrashing
Kmeans Clustering [39]	KM	Thrashing
Symmetric Rank-k [13]	SYRK	Thr.+Conf.
Vector Matrix Multiply [13]	GESUMMV	Thr.+Conf.
MCARLO Pi Estimator [32]	PEst	Friendly
B+tree [3]	B+tree	Friendly
Back Propagation [3]	BP	Friendly
Neural Network [2]	NN	Friendly

3. WORKLOAD CHARACTERIZATION

3.1 Characterization Methodology

In order to understand the cache sensitivity of GPGPU applications, we run our workloads in three different cache scenarios: (1) totally bypassing all memory accesses (i.e. no caches), (2) Bounded caches using the baseline configuration (16KB L1, 786KB L2) and (3) A fully-associative unbounded caches (only cold misses occur in this scenario and it represents the upper bound of performance improvement from using caches). Note that, in unbounded caches, the other cache resources (e.g. MSHRs) are still limited as the baseline configuration and not increased. The results are shown in Figure 2. Further, we analyze and measure the amount of locality that exist in GPGPU workloads in the bounded and unbounded scenarios. We classify locality that occurs in GPPGU application to the following five categories: (1) *Intra-warp* data locality occurs when a cache line is referenced and re-referenced by the same warp. we can further divides the intra-warp locality into two subcategories (intra-thread and inter-thread locality [39]). Intra-thread occurs when a cache line is referenced and re-referenced by the same thread, while inter-thread locality occurs when a cache line is referenced and re-referenced by two different threads within the same warp. (2) *Intra-block* data locality occurs when a cache line is referenced and re-referenced by two different warps within the same thread block. (3) *Intra-core* data locality occurs when a cache line is referenced and re-referenced by two different warps with different thread blocks, and the two thread blocks are assigned to the same core. (4) *Inter-core* data locality occurs when a cache line is referenced and re-referenced by two different warps with different thread blocks, and the two thread blocks are assigned to different cores. Obviously, this locality is only exploited through L2 cache. (5) *Inter-kernel* data locality

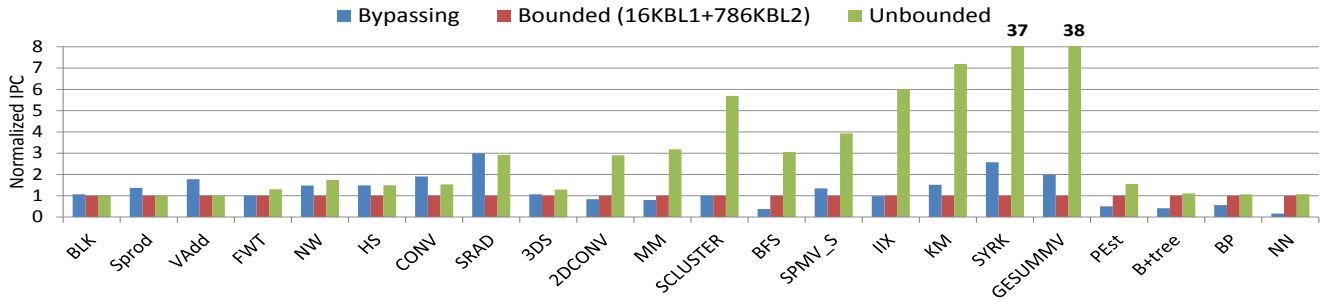


Figure 2: Cache Sensitivity

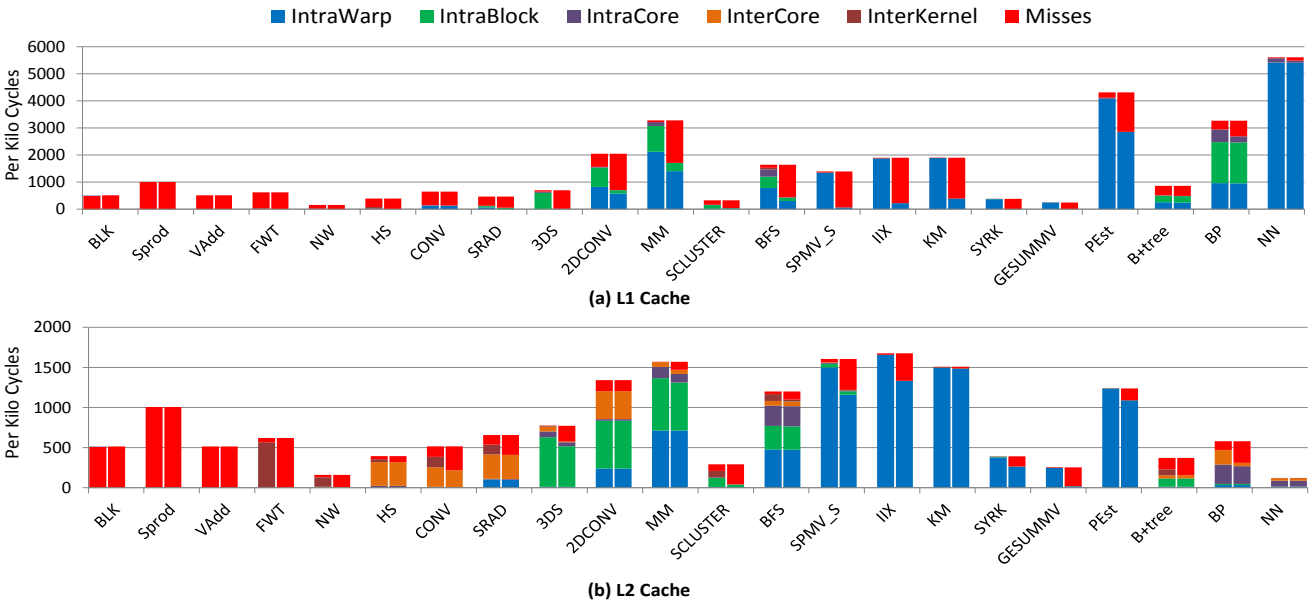


Figure 3: L1/L2 Data Locality Analysis. The left bar represents the locality found in unbounded caches while right bar for bounded caches

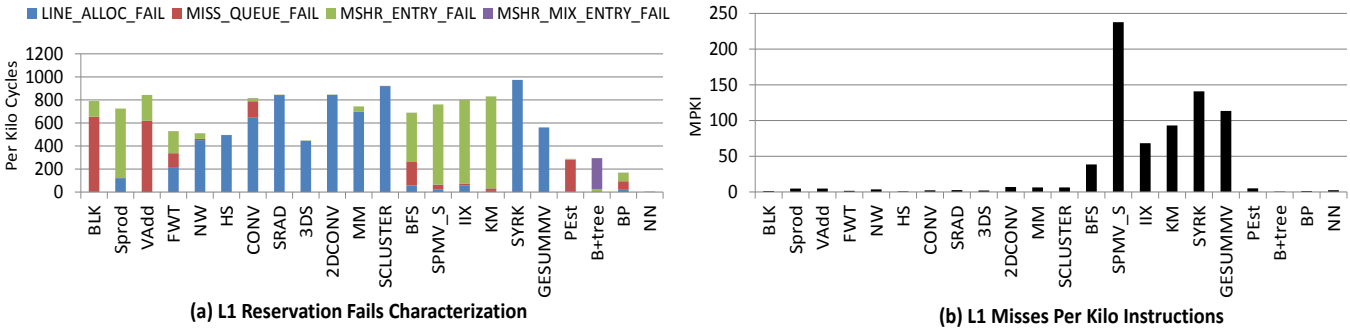


Figure 4: L1 Cache Resource Contention

occurs when a cache line is referenced and re-referenced by two different warps with different kernels. For instance, a data was written by a kernel and a consecutive kernel accesses this data. Figure 3(a) and 3(b) show the amount and type of locality in L1 and L2 respectively. The left bar represents the locality found in unbounded caches while right bar for bounded caches. The L2 cache was evaluated while L1 cache is bounded.

GPGPU workloads exhibit a high level of contention at the few available L1 cache resources (e.g. MHSR entries, cache lines and Miss_Queue entries). When a cache miss occurs, the miss status handling registers (MSHRs) are checked to see whether the same request has already been issued for another warp and is still pending. If the request was found, a MSHR_MIX entry is allocated to ensure the returned request services both warps. If the request wasn't found in MSHRs, an empty MSHR entry is allocated, a cache line is reserved and a read memory request is placed in the Miss_Queue. However, a cache controller may fail to service a miss request due to lack of any requested resource (MSHR_MIX_ENTRY_FAIL, MSHR_ENTRY_FAIL, LINE_ALLOC_FAIL or MISS_QUEUE_FULL). In this case, the memory request causes the pipeline to stall and it will retry the next cycles until all the requested resources are available. Figure 4(a) depicts the L1 reservation fails per kilo cycles for our workloads and Figure 4(b) shows the misses per kilo cycles (MPKI).

3.2 Characterization Results

From experimental results (shown in Figure 2, 3, and 4), we classify our applications into three main categories:

(1) Streaming Applications: We observe that there are applications that don't benefit from caches at all (BLK, Sprod, and VAdd). In these applications, caches hurt the performance instead. As shown in Figure 2, using bounded or unbounded caches leads to loss in performance compared to bypassing. These applications show a streaming behavior in both L1 and L2. They exhibit a high miss rate (up to 99%) in bounded and unbounded caches. For these applications, it is better to bypass their memory accesses since they don't benefit from caches and cause useless unintended contention at L1 (see Figure 4(a)). For (FWT and NW), the unbounded cache is better than bounded and bypassing. These applications also show streaming behavior in bounded and unbounded L1 cache. In contrast, they are full of inter-kernel locality at unbounded L2, however bounded L2 is not large enough to cache the data transferred between kernels. In this work, we bypass these applications, and we leave improving L2 cache to exploit inter-kernel locality for future work. For (HS and CONV), they show a streaming behavior in L1, while they exhibit a high hit rate at L2. They are full of inter-core locality and thus they are L2 cache sensitive. For these applications, it is better to bypass L1 cache only. If we inspect the code of these workloads, we will find that they rely on the on-chip scratchpad memory to exploit locality. Thus, when they are cached, they show streaming behavior in L1. It is worthwhile to note that, an application that uses scratchpad, doesn't necessarily show streaming behavior in L1. For instance, BP relies on scratchpad, however it is full of locality.

(2) Cache Contention Applications: For these workloads, the unbounded cache is better than bounded and bypassing by an order of magnitude (as shown in Figure 2 for SCLUSTER, IIX, SYRK and others). These workloads are full of data locality at the unbounded L1 (as shown in Figure 3(a)), however the limited size of bounded L1 cache and the large number of threads GPGPU executes concurrently makes it susceptible to conflict and capacity misses [17, 18].

Conflict misses mainly occur when a group of warps (Inter-warp conflict contention) or a group of threads within the same warp (Intra-warp conflict contention [18]) access the same cache set within a short period of time. The warps/threads compete on the few available cache lines in cache set (typically between 4-6 lines [29]). Consequently, it causes a high level of LINE_ALLOC_FAIL stalls (associativity stalls). Increasing the associativity of L1 cache is able to alleviate this type of contention [18]. Capacity misses mainly occur when the cache footprint per-warp is large (Inter-warp capacity contention). In this case, with no conflict contention and L1 cache cannot fit all the running warps working set, the warps will compete on the cache lines and the cache is susceptible to thrashing. Cache thrashing causes a high level of MSHR_ENTRY_FAIL stalls and MPKI. Increasing the L1 cache capacity is able to alleviate this type of contention [18]. It has been shown that the code style has an effective role to alleviate/increase cache contention [18]. Writing a highly-divergent non-optimized code (mainly programs that contain loops) may cause a severe cache contention [40].

The applications (SRAD, 3DS, 2DCONV, MM, and SCLUSTER) suffer from inter-warp conflict contention. As shown in Figure 3, they are full of intra-warp and intra-block locality. However, the bounded L1 is not able to exploit these localities, especially intra-block locality. Figure 4(a) illustrates that these applications exhibit a high level of LINE_ALLOC_FAIL stalls, which means that inter-warp conflict contention occurs in these workloads. The applications (BFS, SPMV, IIX, and KM) suffer from inter-warp capacity contention. They contain a large intra-warp locality. Most of these locality are intra-thread (not shown in figure). The running warps evict the cache lines of each other's and cause severe thrashing at L1 (see Figure 3(a)) and high levels of MSHR_ENTRY_FAIL stalls (see Figure 4(a)). Figure 4(b) shows that these thrashing applications exhibit a high level of MPKI over other applications. Hence, MPKI can be used as a good measure to detect thrashing. The applications (SYRK and GESUMMV) suffer from intra-warp conflict contention and inter-warp capacity contention. The LINE_ALLOC_FAIL stalls and MPKI for these applications are high. In section 4.3, we discuss the behavior of these workloads in details. Note that, in thrashing applications, the L2 cache backs up L1 evicted cache lines for future reuse, except for GESUMMV. In GESUMMV, the cache footprint per-warp is large to the extent that thrashing also occurs at L2 (see Figure 3(b)).

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

4. EFFICIENT UTILIZATION OF GPGPU CACHES

In this section we describe our proposed methods to bypass streaming behavior, alleviate thrashing and avoid conflicts.

4.1 Dynamically Bypassing Streaming Applications

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

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

Prior work [39, 40] proposed warp throttling as an effective method to alleviate the cache thrashing problem. The number of active running warps per core is throttled to a lower number such that their cache footprint can be fitted in cache. Static Warp Throttling (SWT) (a.k.a. Best Static Warp Limiting [39]) statically runs an exhaustive search to find the best number of active warps that achieves the highest performance. All possible warp numbers per warp scheduler (24 to 1 in our case) were tested and the best performing one is selected. In contrast, Dynamic Warp Throttling (DWT) aims to find the best number of active warps dynamically by hardware. Cache Conscious Wavefront Scheduling (CCWS) [39] and Divergence-Aware Warp Scheduling (DAWS) [40] are two proposed schemes to implement DWT. CCWS uses a victim tag array, called lost locality detector, to detect warps that have lost locality due to thrashing. These warps are prioritized till they exploit their locality while other warps are descheduled (not allowed to issue any load instructions). DAWS introduced a divergence-based cache footprint predictor to estimate the amount of locality in loops required by each warp. DAWS uses these predictions to prioritize a group of warps such that the cache footprint of these warps don't exceed the capacity of the L1 cache.

CCWS and DAWS have a fine-grained control on warp throttling (i.e. the number of active warps is variable over time depending on the thrashing level). This gives them an advantage over SWT. However, it has been shown that the coarse-grained SWT approach has a comparable speedup to CCWS and DAWS. SWT is able to outperform CCWS and almost achieves the same performance of DAWS on average [39]. This is due to the fact that many GPGPU applications show a consistent thrashing level over time. In addition, SWT tries to find the best trade-off number of warps that works well at different thrashing levels and improves the overall performance. However, SWT is not a practical solution. The programmer needs to do an exhaustive search for each application. Moreover, SWT is input sensitive, which means that the best number of warps changes when running the same application with different input sets [39]. In this work, we propose Dynamic Warp Throttling via Cores Sampling (DWT-CS). DWT-CS uses a similar approach as SWT (i.e. exhaustive searching), however it lets the hardware handle the searching process. Thus, DWT-CS overcomes SWT shortcomings. The idea of core sampling was proposed by Lee et al. [25] by applying different cache management policies to different cores and collecting samples to see how these policies behave. In this work, we employ a similar mechanism to find the best number of active warps that alleviates thrashing and efficiently utilizes L1 cache.

Table 3: The number of active warps (per warp scheduler) achieved by SWT vs DWT-CS

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

DWT-CS monitors the MPKI at L1 over sampling periods (from Figure 4(b), L1 MPKI can be used as a good measure to detect thrashing). At the end of each sampling period, it checks whether the MPKI has exceeded a specific threshold for N consecutive periods. If so, all GPU cores are sampled with different number of active warps, equals to the core ID (For example, core#1 throttles the active warps to only one warp, core#2: two active warps and so on). After M sampling periods, all cores send the number of instructions committed during the sampling periods to the coordinator core (for example, the middle core, core#8 in our case). The coordinator core finds the core ID (i.e. number of warps) that has executed the maximum number of instructions. The winner core ID is propagated to all the cores. Next, the cores throttle the number of active warps to the new propagated value and it remains till the end of kernel execution. In case, the winner core is the last core (core# 15 in our case), another sampling period is relaunched to test the other numbers of warps (16-24). However, this case did not occur in our benchmarks and it rarely exists. When the same kernel is relaunched and MPKI exceeds the threshold, it doesn't sample the cores again. Instead, it uses the same number of warps obtained before.

Table 3 shows the best number of active warps achieved by SWT and DWT-CS. The DWT-CS is able to achieve the same number of warps as SWT for all benchmarks except for BFS which consists of small kernels and suffers from phased execution (i.e. non-steady thrashing level).

Our proposed DWT-CS is a cost-effective method. It is able to slightly outperform CCWS on average over thrashing applications (see Section 5), while requiring negligible hardware overhead. CCWS needs extra hardware (victim tag arrays) to detect thrashing, whereas DWT-CS needs only two counters to calculate the committed instructions and cache misses in order to measure the MPKI over sampling periods. A few registers are also needed to save the best number of warps per-kernel for future reuse. Moreover, the coordinator core can use one of the built-in SIMD units that support MAX instruction [31] to find the winner core ID. DWT-CS takes around 60K cycles to detect thrashing and find the optimal number of warps. In real applications, this overhead time can be neglected.

4.3 Pseudo Random Interleaved Cache (PRIC)

The problem of CPU cache associativity has been widely studied in literature. Many works proposed different techniques to improve the cache indexer function that is responsible for interleaving memory accesses over cache sets. Prime modulo interleaving [24, 22], 1-skew storage [15], logical data skewing [41], Xor-based functions [42] and Pseudo Random Interleaving [38] have been proposed, instead of the conventional sequential interleaving, in an attempt to improve cache associativity and avoid conflicts. It has been shown that the Pseudo Random Interleaving Cache (PRIC) is a cost-effective high-performance approach [38, 12, 43]. In this work, we employ PRIC for GPU caches to alleviate associativity stalls and eliminate conflict misses.

In a sequential interleaving cache consisting of $M = 2^m$ cache sets and a cache line size $B = 2^b$ bytes, a N-bit memory location whose address is A[N-1:0], has cache index of A[m+b-1:b] and a Tag address A[N-1:m+b]. Figure 5(a) depicts a simple example of how memory locations are interleaved over cache sets in case of sequential interleaving. In an application that generates a stream of M memory references in a short period of time, with an access stride S, has a n-way conflict degree where n=M/gcd(M,S), and gcd stands for the greatest common divisor. From this equation, we can observe that even strides will cause a high level of conflict degree. For example, assume the sequential memory interleaving shown in Figure 5(a), a reference stream with an access stride of 2 (i.e. 0,2,4,6,8,10,12,14) has a 4-way conflict degree (i.e. all the memory references will be mapped to only 4 cache sets out of 8). Each pair of the addresses (0,8), (2,10), (4,12), (6,14) will map to the same cache set and may cause associativity stalls (in case the cache set contains cache lines less than mapped memory references). The worst case scenario occurs when the reference sequence has a stride which is a multiple of M, thus causing a 1-way conflict where all references map to the same cache set. On the other hand, all odd strides have no common divisor with M greater than one (recall that M is a power-of-2 number) and hence they don't cause any conflicts and the memory references will be distributed evenly over the cache sets. However, it is important to note that even strides, especially strides that are of multiple M, occur frequently in GPGPU applications. For instance, Figure 6 shows a GPGPU frequent scenario to access a 2-D matrix. In SYRK, the output element (i,j) is calculated by multiplying row(i) by row(j) of matrix A. The A matrix rows are aligned to the cache line size (i.e. row size=K*line size) and are stored in a row-major order form in the main memory. On each loop iteration, each 32 threads within a warp read 32 elements from different consecutive rows of matrix A. When K is a multiple of the number of cache sets (32 in our baseline), all the memory reference loads will map to the same cache set causing a high level of associativity stalls and conflict misses. A matrix whose row size equals 32*n*line size, where n>=1, frequently exists in GPGPU applications.

In PRIC, as shown in Figure 5(b), each M consecutive memory locations have different permutation over the cache sets in a way that make them near-randomly interleaved. This near-random interleaving makes PRIC resistant to all strides, especially strides that are multiple of M. PRIC is based on Polynomial Modulus Mapping in which the memory location, A, is expressed as a polynomial function whose coefficients are in the Galios GF(2). For example, memory location 21 is expressed as $(x^4 + x^2 + 1)$. Let P(x) be a polynomial of order m, and A(x) be the polynomial of order N that is associated with memory location A. Then A(x) can be uniquely represented as A(x) = V(x)*P(x) + R(x) where V(x) and R(X) are polynomials over GF(2) and R(x) is of order less than m. V(x) and R(x) can be seen as the polynomial representation of the corresponding tag and cache index. Hence, the cache index of address R(x) = A(x) mod P(x). It was found that for the best performance and permutation, P(x) should be an Irreducible Polynomial Function (I-Poly). P(x) is said to be irreducible if no two non constant polynomials g(x) and h(x) with rational coefficients such that P(x)=g(x)*h(x) exists [28]. Rau [38] shows how the computation of cache index R(x)=A(x) mod P(x) can be carried out by the vector-matrix product of the address and a matrix of single-bit coefficients, named H-matrix (i.e. I[m-1:0] = A[N-1:b]*H-matrix). In GF(2), multiplication and addition are equivalent to AND and XOR boolean function, and if the matrix is constant, the AND gates can be omitted and the mapping then requires just XOR gates with fan-in from 2 to n [12].

In our case, the baseline has $32 = 2^5$ cache sets, thus m=5. There are six irreducible polynomial functions of degree 5 over GF(2) [28] and they are Poly (37, 41, 47, 55, 59, 61). In this study, we use Poly(37). The corresponding Xor-ing boolean equations of Poly(37) is listed in Figure 7. As shown in the figure, the cache index I[4:0] is generated by Xor-ing some bits of the memory address A[31:0]. For more information on the proofs and theorems behind the polynomial modulus, as well as the method to generate the H-matrix and Xor-ing equations, kindly refer to [38].

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

(a) Sequential Interleaving

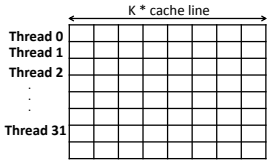
S0	S1	S2	S3	S4	S5	S6	S7
0	4	2	6	1	5	3	7
13	9	15	11	12	8	14	10
23	19	21	17	22	18	20	16
26	30	24	28	27	31	25	29
35	39	33	37	34	38	32	36
46	42	44	40	47	43	45	41
52	48	54	50	53	49	55	51
57	61	59	63	56	60	58	62

(b) Pseudo Random Interleaving

Figure 5: Memory locations interleaving over cache sets (Assume 6-bit memory address, 1-byte cache line and 8 cache sets)

```
//from SYRK
//element c(i,j) = row i*row j
int j = blockIdx.x * blockDim.x + threadIdx;
int i = blockIdx.y * blockDim.y + threadIdx;
int k;
for(k=0; k< M; k++)
{
  c[i * N + j] += alpha * a[i * M + k] * a[j * M + k];
}
```

(a) SYRK



(b) A Matrix

Figure 6: An example of 1-way conflict degree in SYRK workload. When K is multiple of the number of cache sets, all 32 threads will map to the same cache set

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

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

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

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

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

Figure 7: The Xor-ing equations corresponding to Poly(37) that we use in the study

Recent works [45, 27, 18] proposed cache bypassing on associativity stalls. However, these methods are not effective to efficiently utilize cache resources. In many cases, bypassing occurs while the other cache sets are underutilized. For instance, Memory Request Prioritization Buffer (MRPB) [18] allows memory request that encounters associativity stall (i.e. LINE_ALLOC_FAIL) to bypass L1 cache. In SYRK workload shown in Figure 6(a), when all the 32 threads map to the same cache set, only the first four threads will successfully allocate a cache line (assume 4-way associativity) and the remaining 28 threads will bypass the L1 cache. This is because all the lines within the cache set will be reserved by the first four threads. However, our empirical search shows that the other cache sets are underutilized and thus it is better to distribute the remaining threads over the underutilized cache sets instead of bypassing. Moreover, on the second iteration, the same 32 threads access the second elements from the matrix rows and they will map to the same cache set again. The first four threads hit the cache, while the next four threads cause miss and consequently they evict the previous threads cache lines. This behavior is repeated over the next loop iterations, the first four threads and the second four threads evict the lines of each other. This behavior causes severe conflict misses for SYRK and GESUMMV as shown in Figure 3(a). Hence, cache bypassing is not an efficient method to handle the GPU cache associativity problem.

5. EXPERIMENTAL RESULTS

We compare DWT-PRIC (i.e. bypassing streaming + DWT-CS + PRIC) to previously proposed CCWS [39] and MRPB [18]. We have discussed CCWS in section 4.2. CCWS addresses the thrashing problem only and doesn't consider conflict contention. On the other hand, MRPB employs two techniques to alleviate the thrashing and conflict problems. First, a FIFO requests buffer is used to reorder memory references so that requests from the same warp are grouped and sent to the cache together and thus reducing the number of warps that access the cache at a time. Second, MRPB allows memory request that encounters associativity stall to bypass L1 cache. We have seen in section 4.3 that bypassing strategy is not effective to handle cache associativity. Table 4 presents the configuration parameters used for CCWS, MRPB and DWT-PRIC. In CCWS, the value Kthrottle was tuned to our baseline architecture by the same way described in [39]. In MRPB, we use the same configuration in [18] that achieved the highest performance. In DWT-PRIC, the configuration parameters were selected based on empirical analysis.

Table 4: CCWS/MRPB/DWT-PRIC Configuration

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

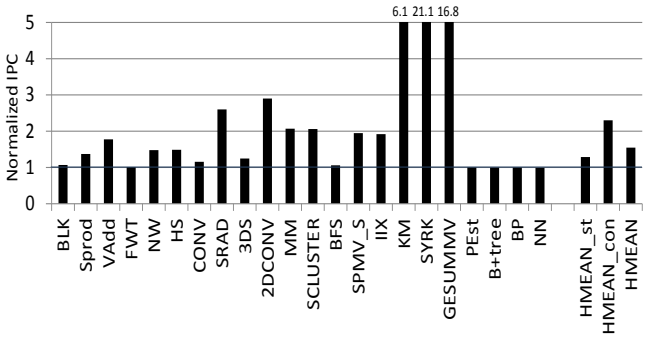


Figure 8: Performance Improvement on all benchmarks normalized to baseline

Figure 8 presents the performance improvement (Instruction Per Cycle) of our proposed methods (DWT-PRIC) on all benchmarks with respect to baseline. DWT-PRIC achieves a harmonic mean 1.54X performance improvement over baseline. It improves the average performance of streaming and contention applications by 1.2X and 2.3X respectively. Some applications exhibit an improvement up to 21X (SYRK) and 16.8X (GESUMMV). In addition, it doesn't cause any performance degradation in the cache friendly applications.

Figure 9 illustrates the DWT-PRIC performance improvement on cache contention applications with respect to baseline, CCWS and MRPB. In total, DWT-PRIC outperforms CCWS and MRPB by a harmonic mean 1.7X and 1.5X respectively. For inter-warp conflict contention applications, DWT-PRIC improves performance by a harmonic mean 2.1X and 1.37X over CCWS and MRPB respectively, due to the efficiency of PRIC in utilizing cache sets. For inter-warp thrashing applications, DWT-PRIC results in a harmonic mean 1.02X and 1.25X performance improvement over CCWS and MRPB respectively. DWT-PRIC shows significant improvement on the applications that show consistent thrashing level and coherent control flow divergence (IIX and KM). On the other hand, CCWS slightly performs better in SPMV and BFS due to the unsteady thrashing level of these appli-

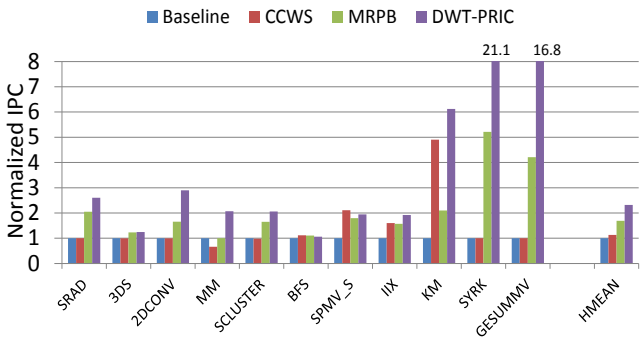


Figure 9: Performance Improvement on Contention Applications compared to CCWS and MRPB

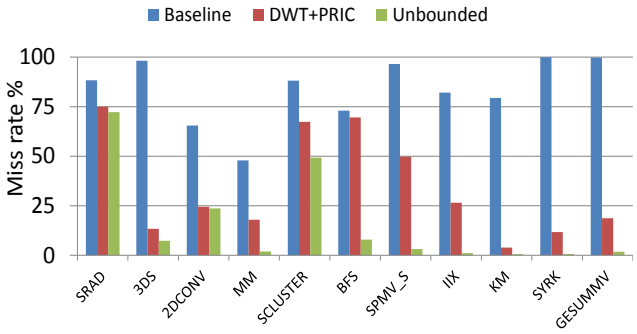


Figure 10: L1 Miss Rate Reduction

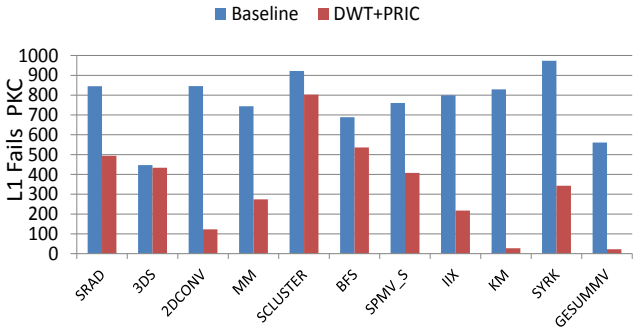


Figure 11: L1 Reservation Fails Reduction

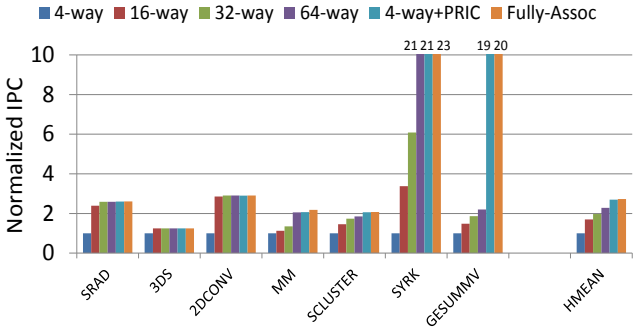


Figure 12: PRIC compared with High Associativity

cations. SPMV and BFS are highly control flow divergent and thus the per-warp cache footprint changes over time depending on warp's active mask. DAWS [40] is a divergence aware warp throttling mechanism that can improve the performance of these applications further. For applications that exhibit both intra-warp conflict contention and inter-warp thrashing, DWT-PRIC achieves a superior performance improvement and outperforms CCWS and MRPB by a harmonic mean 18X and 4X respectively. These workloads benefit from both DWT-CS and PRIC. Neither DWT-CS alone nor PRIC alone is able to improve the performance of these applications.

Figure 10 shows the reduction in L1 miss rate for DWT-PRIC and unbounded cache compared to the baseline. More than an 80% reduction is observed under DWT-PRIC for 3DS, KMN, SYRK and GESUMMV. Also, a miss rate reduction up to 60% is noticed for 2DCONV, MM and IIX. Moreover, DWT-PRIC nearly achieves the same miss rate of unbounded cache for SRAD, 3DS, 2DCONV and KM.

Figure 11 shows the reduction in L1 reservation fails per kilo cycles for DWT-PRIC compared to the baseline. DWT-PRIC reduces the reservation fails by 20% on average. Applications such as 2DCONV, KM, and GESUMMV show a significant reduction (up to 90%). MM, IIX, and SYRK also show a considerable reduction (up to 60%). In contrast, SCLUSTER and SRAD still show a high level of fails PKC, especially MSHR_ENTRY_FAIL, due to the noticeable streaming behavior of these application (up to 50% and 70% miss rate in unbounded cache, as shown in Figure 10).

Increasing the associativity of L1 cache is a straightforward approach to mitigate associativity stalls and conflict misses. For instance, recent AMD's Graphics Core Next (GCN) GPUs [1] use 64-way associativity for their 16KB L1 caches. Figure 12 presents PRIC performance improvement on inter-warp and intra-warp conflict contention applications with respect to high associativity and fully-associative caches. In all cases, the L1 cache capacity is fixed and we assume idealistic 1-cycle hit latency. PRIC with 4-way associativity outperforms the 16-way, 32-way, and 64-way caches by a harmonic mean 1.6X, 1.4X, and 1.16X respectively. Moreover, PRIC is able to achieve 97% of fully-associative cache's performance. In addition to that, increasing associativity requires a considerable hardware overhead (tag comparators and large data selectors), which increasing both access latency and power consumption [37].

6. RELATED WORK

Different methods have been proposed in literature to alleviate the problems associated with GPU caches. Table 5 summarizes and compares between these works in terms of thrashing detection, thrashing handling and conflict stalls mitigation. We can classify these works to the following:

(1) CTA throttling:

Jog et al. [19] proposed CTA-aware-locality scheduling that gives a group of CTAs higher priority to keep their data in the L1 cache such that they get the opportunity to reuse it. Kayiran et al. [20] proposed dynamic CTA scheduling, which attempts to allocate optimal number of CTAs per-core in order to reduce contention in the memory subsystem. Lee et al. [26] explored two alternative thread block scheduling schemes. Lazy CTA scheduling was proposed to leverage GTO scheduler to determine the optimal number of

Table 5: Related GPU Cache Contention Works

	Thrashing Detection	Thrashing Handling	Conflict stalls
OWL [19]	–	CTA throttling	–
Dynamic CTA [20]	memory latency	CTA throttling	–
Lazy CTA [26]	#Instructions issued under GTO	CTA throttling	–
CCWS [39]	Victim cache	Warp throttling	–
DAWS [40]	L1 footprint prediction	Warp throttling	–
PCAL [27]	–	Warp throttling (CCWS)	Warp Bypassing
CCA [45]	L1 footprint prediction	Warp throttling (DAWS)	Warp Bypassing
MRPB [18]	–	Memory request prioritization	Bypassing on stalls
G-Cache [7]	Victims bits at L2	adaptive bypass policy	–
CBWT [6]	Victims bits at L2	PDP cache management	–
DWT+PRIC	L1 MPKI	Warp throttling (DWT-CS)	PRIC

CTAs per core. They also showed how block CTA scheduling (BCS), where consecutive thread blocks are assigned to the same cores, can exploit inter-block locality (i.e. intra-core and inter-core locality). It is obvious that fine-grained warp throttling mechanisms, such as DWT-CS, are better than coarse-grained CTA throttling mechanisms. Based on an experiment (not shown here), static warp throttling outperforms static CTA throttling by 2X on average.

(2) Warp Throttling: In addition to the previously discussed CCWS and DAWS, other recent works were proposed to improve warp throttling. Li [27] observed that throttling techniques leave memory bandwidth and other chip resources (L2 cache, NOC and EUs) significantly underutilized. Thus, he proposed a cache bypassing scheme on top of CCWS, called Priority-based Cache Allocation (PCAL). PCAL starts from an optimal number of active warps, that alleviates thrashing and conflicts, then extra inactive warps are allowed to bypass cache and utilize the other on-chip resources. Thus, PCAL reduces the cache thrashing and effectively employs the chip resources that would otherwise go unused by a pure thread throttling approach. A similar approach was proposed by Zheng et al. [45], called Adaptive Cache and Concurrency (CCA). CCA improves DAWS by allowing extra inactive warps and some streaming memory instructions from the active warps to bypass the L1 cache and utilize on-chip resources.

However, PCAL and CCA employ bypassing while leaving cache sets underutilized. For example, recall the SYRK example in section 4.3, PCAL throttles the number of active warps that can access cache to only one warp and allows two warps to bypass the cache in an attempt to utilize chip resources. However, as we have seen, the cached warp only utilizes one cache set, moreover it utilizes it in an inefficient manner (the threads map to the same set causing severe associativity stalls and conflict misses). In contrast, DWT-PRIC effectively utilizes cache sets by allowing two warps to access cache and fairly distributing their memory requests over sets. Note that, PCAL or CCA can be employed on top of our DWT-PRIC for further performance improvement and efficient utilization of L1 cache sets as well as on-chip resources.

(3) MRPB:
We have discussed MRPB in section 5. MRPB proposed FIFO buffers to prioritize memory requests that are generated by the same warp. It also proposed cache bypassing on associativity stalls and we have shown that the MRPB bypassing mechanism is not an effective method.

(4) Cache Replacement Policy:
Chen et al. [7] proposed G-Cache to alleviate cache thrashing. To detect thrashing, the tag array of L2 cache is enhanced with extra bits (victim bits) to provide L1 cache by some information about the hot lines that have been evicted before. An adaptive cache replacement policy is used by L1 cache to protect these hot lines. Chen [6] continued his work and proposed Coordinated Bypassing and Warp Throttling (CBWT). CBWT adopts a thrashing-resistant CPU cache management scheme, Protection Distance Prediction (PDP) [10], to GPU cache. PDP employs cache bypassing to enable protection on hot cache lines and thus alleviate cache thrashing. Excessive bypassing may over-saturate the on-chip network. Therefore, cache bypassing policy is coordinated with a dynamic warp throttling mechanism to avoid over-saturating on-chip resources. However, the previous works don't address the associativity problem and they employ cache replacement policy to alleviate thrashing, while we use warp throttling mechanism.

7. CONCLUSION AND FUTURE WORK

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

Our proposed method improves the average performance of streaming and contention applications by 1.2X and 2.3X respectively. Compared to prior work, it achieves 1.7X and 1.5X performance improvement over CCWS and MRPB respectively. Moving forward, we plan to explore a more sophisticated fine-grained bypassing mechanism. We also plan to improve our DWT-CS to be more resistant to unstable thrashing applications.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and Mohamed Hammad for their insightful feedback on this paper. We also thank Wenhao Jia and Tim Rogers for generously sharing the source code of MRPB and CCWS respectively. Special thanks go to Ahmed ElTantawy for his assistance with GPGPU-sim tool.

9. REFERENCES

[1] AMD. AMD's Graphics Core Next Arhcitecure whitepaper.

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

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

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

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

[6] X. Chen, L.-W. Chang, C. I. Rodrigues, L. Ji, Z. Wang, and W. mei Hwu. Adaptive Cache Management for Energy-efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

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

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

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

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

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

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

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

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

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

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

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

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

[19] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 395–406. ACM, 2013.

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

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

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

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

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

[25] J. Lee and H. Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012.

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

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

[28] Mathworld. mathworld.wolfram.com/IrreduciblePolynomial.html.

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

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

[31] NVIDIA. CUDA C Programming Guide v5.5.

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

[33] NVIDIA. NVIDIA Next Generation CUDA Compute Architecture: Kepler GK110.

[34] OpenCL. The OpenCL Specification version 2.0. http://www.khronos.org.

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

[36] Paulius Micikevicius. GPU Performance Analysis and Optimization, 2012.

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

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

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

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

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

[42] G. S. Sohi. Logical data skewing schemes for interleaved memories in vector processors. 1988.

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

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

[45] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive Cache and Concurrency Allocation on GPGPUs. *Computer Architecture Letters*, 2014.