SACAT: Streaming-Aware Conflict-Avoiding Thrashing-Resistant GPGPU Cache Management Scheme

Mahmoud Khairy, Mohamed Zahran, *Senior Member, IEEE*, and Amr Wassal, *Member, IEEE*

Abstract—Modern graphical processing units (GPUs) are equipped with general-purpose L1 and L2 caches to reduce the memory bandwidth demand and improve the performance of some irregular general-purpose GPU (GPGPU) applications. However, due to the massive multithreading, GPGPU caches suffer from severe resource contention and low data-sharing which may lead to performance degradation instead. This paper proposes a low-cost streaming-aware conflict-avoiding thrashing-resistant (SACAT) GPGPU cache management scheme that efficiently utilizes the GPGPU cache resources and addresses all the problems associated with GPGPU caches. The proposed scheme employs three orthogonal techniques. First, it dynamically detects and bypasses streaming applications at fine granularity. Second, a dynamic warp throttling via cores sampling (DWT-CS) is proposed to alleviate cache thrashing. DWT-CS runs an exhaustive search over cores to find the best number of warps that achieves the highest performance. Third, it employs pseudo random interleaving cache (PRIC), which is an improved cache indexing function based on polynomial modulus mapping, to mitigate associativity stalls and eliminate conflict misses. Experimental results demonstrate that the proposed scheme achieves a 1.87× and a 1.5× performance improvement over the cache-conscious wavefront scheduler (CCWS) and the memory request prioritization buffer (MRPB), respectively.

Index Terms—Cache management, GPGPU

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 [1], [2]. However, GPGPU programming is a difficult task. The programmer has to explicitly manage the on-chip scratchpad memory to generate coalesced memory accesses and utilize data locality [2], [3]. Moreover, it has been shown that the memory throughput has become a limiting factor for the performance of many GPGPU applications [1]. To address these issues, modern GPUs [4], [5] are equipped with a general purpose on-chip cache hierarchy in an attempt to reduce the off-chip memory bandwidth demand, increase the 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 a GPU executes concurrently. The recent NVIDIA's Fermi GPU [4] supports 1,536 active threads per core, and the L1 cache size is configurable to 16 or 48 KB. 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 the NVIDIA's Kepler GPU [5] which has 2,048 active threads per core and a read-only L1 data cache of size 48 KB. This indicates that the GPU cache is not designed to keep the per-thread working set, as in CPUs. For example, the Intel core i7 CPU [6] contains 2 threads and a 32 KB L1 per core, i.e., 16 KB per thread. In fact, GPU caches were designed to make use of some access patterns that exhibit a small cache footprint per thread, and can fit in the cache (e.g., spilled registers, small-stride access pattern [3] and inter-core data locality [5]). Therefore, when GPGPU applications with a large cache footprint per-thread rely on caches to make use of data locality, the active threads will compete for the few available cache lines and the L1 cache will be susceptible to *thrashing*. Moreover, the limited number of set associativity, typically between 4 and 6 [7], makes the L1 cache more vulnerable to *associativity* stalls and *conflict* misses. Additionally, GPGPU applications which use the scratchpad memory, to make use of locality, exhibit a *streaming* behavior on the L1 cache. Cache management schemes which are unaware of these streaming applications lead to useless unintended contention at the L1 cache, which in turn may cause performance degradation.

Different techniques have been proposed to alleviate cache thrashing including CTA throttling [8], [9], warp throttling [10], [11], [12], [13], FIFO buffer [14], and thrashing-resistant cache replacement policy [12], [15], [16]. However, many of these techniques address the cache thrashing problem only, incur a considerable storage overhead, and require significant changes to the baseline architecture. On the other hand, cache bypassing [12], [13], [14] was proposed to mitigate the associativity stalls. However, cache

- *M. Khairy and A. Wassal are with the Department of Computer Engineering, Cairo University, Giza 12613, Egypt. E-mail: {makhairy, wassal}@eng.cu.edu.eg.*
- *M. Zahran is with the Department of Computer Science, New York University, New York, NY 10003. E-mail: mzahran@cs.nyu.edu.*

Manuscript received 8 Nov. 2015; revised 1 Nov. 2016; accepted 5 Nov. 2016. Date of publication 15 Nov. 2016; date of current version 17 May 2017. Recommended for acceptance by M. Ripeanu. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2016.2627560

bypassing does not utilize the available cache resources efficiently. In many cases, bypassing occurs while cache sets are underutilized.

This paper proposes a low-cost streaming-aware conflict-avoiding thrashing-resistant (SACAT) GPGPU cache management scheme that efficiently utilizes the GPGPU cache resources. The proposed scheme employs three techniques. First, it dynamically detects and bypasses applications that exhibit a streaming behavior in L1 or L2 cache at fine granularity. Second, a dynamic warp throttling via cores sampling (DWT-CS) technique is proposed to alleviate cache thrashing. DWT-CS monitors the MPKI at L1, and when it exceeds a specific threshold, the number of active warps in each GPU core is sampled. Then, the active warps for all cores will be throttled to the number of warps associated with the winner core (the core that achieved the highest performance during the sampling period). Finally, an improved cache indexing function, namely, pseudo random interleaving cache (PRIC) is proposed. It is based on polynomial modulus mapping [17], and is used to mitigate associativity stalls and eliminate conflict misses. PRIC near-randomly and fairly distributes memory accesses over cache sets and thus efficiently utilizes the cache resources. This is the first scheme that addresses three very important GPGPU cache problems including thrashing, associativity contention and streaming behavior. Experimental results demonstrate that the proposed scheme requires simpler hardware and achieves a harmonic mean 1.87× and 1.5× performance improvement over previously proposed cache-conscious wavefront scheduler (CCWS) [10] and the memory request prioritization buffer (MRPB) [14], respectively.

This paper also presents a characterization methodology to analyze and measure the amount of locality that exist in GPGPU workloads by using a fully-associative unbounded cache. The characterization results reveal that many GPGPU applications have large working sets or poor cache reuse and do not benefit from the cache hierarchy. On the other hand, some GPGPU applications exhibit a high level of cache thrashing and/or associativity contention.

The rest of this paper is organized as follows. Section 2 describes the GPGPU programming model, baseline architecture, and simulation environment. Section 3 describes workload characterization methodology. Section 4 describes the proposed cache management scheme, SACAT. Experimental results are presented in Section 5, and a comparison to previous work is presented in Section 6. The conclusions are given in Section 7.

2 GPGPU Programming Model and Baseline Architecture

2.1 GPGPU Programming Model

The CUDA [18] or OpenCL [19] programming model allows 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 three-dimensional scalar threads. Threads within the same thread block communicate with each other through a shared on-chip scratchpad memory and synchronization primitives. During run-time, each consecutive 32 threads are grouped together to formulate a warp, or a wavefront. Warps are executed in a single instruction multiple-threads (SIMT) model. In a 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 technique.

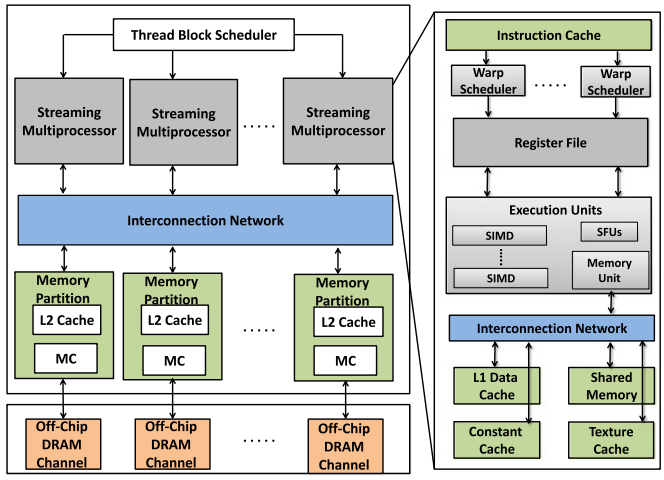


Fig. 1. Baseline GPGPU architecture.

2.2 Baseline Architecture

Fig. 1 illustrates the baseline GPGPU architecture. It consists of multiple GPU cores, called 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, called shared memory. It also contains 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 which are shared among the SMs. The SMs and the memory partitions are connected via an on-chip network.

The thread block scheduler distributes the thread blocks among SMs in a load-balanced round-robin fashion [20]. A thread block is dispatched to an SM only if the required resources of the thread block are available on this SM (e.g., register file, shared memory, warp scheduler entries, etc.). Thread blocks are subdivided by hardware into warps. Each SM contains a number of warp schedulers. 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 [10]. The baseline architecture handles control flow divergence and re-convergence using a post dominator (PDOM) re-convergence stack [21]. Each SM contains a memory-coalescing unit which attempts to coalesce memory requests of active threads within each warp into the fewest possible cache line-sized memory requests.

2.3 Simulation Environment

The baseline architecture is simulated using GPGPU-Sim v3.2.1 [20], a publicly-available cycle-accurate GPGPU simulator. The GPU simulator is configured to be similar to NVIDIA Fermi GTX480 [4], [22]. The configuration file

1. In this paper, we use the terms GPU core and SM interchangeably.

TABLE 1
The Simulated Baseline GPGPU Configuration

SM configuration	15 SMs, 700 MHZ, 1,536 threads, 32 threads/warp, 48 warp/SM, SIMD width = 32
L1 Cache	16 KB/4-way associativity/128 B cache line/32 MSHR entries
L2 Cache	6 partitions x 128 KB/16-way/128 B cache line/ 32 MSHR entries
# Warp schedulers	2 per SM (24 warps per scheduler)
Warp scheduling	Greedy-then-oldest (GTO) [10]
Memory Model	6 GDDR5 Memory Controllers (MCs), BW=179.2 GB/s

provided with GPGPU-Sim is used without any modifications. The configuration parameters are listed in Table 1. MSHR refers to the miss status handling register. The simulator was modified to implement the proposed techniques, in order to evaluate them.

A wide range of GPGPU CUDA workloads was considered, including applications from Rodinia [23], Poly-Bench [24], and NVIDIA SDK [25]. NN, IIX, SPMV_S, and KM are adopted from GPGPU-sim workloads [20], MapReduce [26], SHOC [27], and CCWS applications suite [10], respectively. Table 2 lists all of the 22 applications[2] used. The table also shows the characterizations of these applications based on contention type and streaming behavior. For more details about this characterization methodology, see the following section. In our experiments, all applications run until completion, except for SYRK, GESUMMV, and SCLUSTER, due to their long simulation times. More specifically, SYRK and GESUMMV run only up to 100 million instructions, while SCLUSTER runs up to 300 million instructions.

3 Workload Characterization

3.1 Characterization Methodology

In order to assess the cache sensitivity of GPGPU applications, the selected applications listed in Table 2 are run using three different cache scenarios:

1) Totally bypassing all memory accesses, i.e., no caches.
2) Bounded caches using the baseline configuration (16 KB L1, 786 KB L2)
3) Fully-associative unbounded caches. Only cold misses occur in this scenario. Therefore, it represents the upper bound on the performance improvement gained from using caches.

Note that, in unbounded caches, the other cache resources (e.g., MSHRs) are still limited to the baseline configuration listed in Table 1. Fig. 2 plots the normalized instruction per cycle (IPC) for each of the simulated applications using the three cache scenarios.

Furthermore, to analyze the locality in GPGPU workloads, the amount and type of locality in each simulated application is measured, in both the bounded and unbounded scenarios. Jia, et al. [14] found that there are different types of locality existing in GPGPU workloads. In this paper, locality is classified into two main categories:

2. In this paper, the words workloads and applications are used interchangeably.

TABLE 2
The Simulated GPGPU Workloads

Name	Abbrev.	Contention Type + Streaming behavior
Black Scholes [25]	BLK	Fully L1/L2
Scalar Product [25]	Sprod	Fully L1/L2
Vector Addition [25]	VAdd	Fully L1/L2
Fast Walsh Transform [25]	FWT	Fully L1
Needleman-Wunsch [23]	NW	Fully L1
Hot Spot [23]	HS	Fully L1
Separable Convolution [25]	CONV	Fully L1
Structured grid [23]	SRAD	XW-Conflict+Semi
3D Stencil [23]	3DS	XW-Conflict
2D Convolution [24]	2DCONV	XW-Conflict
2 Matrix Multiplication [24]	MM	XW-Conflict
Stream Cluster [23]	SCLUSTER	XW-Conflict+Semi
Breadth First Search [23]	BFS	XW-Capacity+Semi
Sparse Matrix Vec. Mult. [27]	SpMV	XW-Capacity
Inverted Index [26]	IIX	XW-Capacity+Semi
Kmeans Clustering [10]	KM	XW-Capacity
Symmetric Rank-k [24]	SYRK	XW-Cap+IW-Cont
Vector Matrix Multiply [24]	GESUMMV	XW-Cap+IW-Cont
MCARLO Pi Estimator [25]	PEst	Friendly
B+tree [23]	B+tree	Friendly
Back Propagation [23]	BP	Friendly
Neural Network [20]	NN	Friendly

1) *Intra-warp* data locality occurs when a cache line is referenced and re-referenced by the same warp. Intra-warp locality can be divided further into two subcategories [10]:
   a) *Intra-thread* data locality occurs when a cache line is referenced and re-referenced by the same thread.
   b) *Inter-thread* data locality occurs when a cache line is referenced and re-referenced by two different threads within the same warp.
2) *Inter-warp* data locality occurs when a cache line is referenced and re-referenced by two different warps. Inter-warp locality can be divided further into four subcategories:
   a) *Intra-block* data locality occurs when a cache line is referenced and re-referenced by two different warps within the same thread block.
   b) *Intra-core* data locality occurs when a cache line is referenced and re-referenced by two different warps within different thread blocks, and the two thread blocks are assigned to the same core.
   c) *Inter-core* data locality occurs when a cache line is referenced and re-referenced by two different warps within different thread blocks, and the two thread blocks are assigned to different cores. Obviously, this locality is utilized only through L2 cache [28].
   d) *Inter-kernel* data locality occurs when a cache line is referenced and re-referenced by two different warps within different kernels, e.g., a data item was written by a kernel and a consecutive kernel accesses this data.

Fig. 3 plots the amount and type of locality in L1 and L2 for all applications. The left bar represents the locality found in unbounded caches, whereas the right bar represents

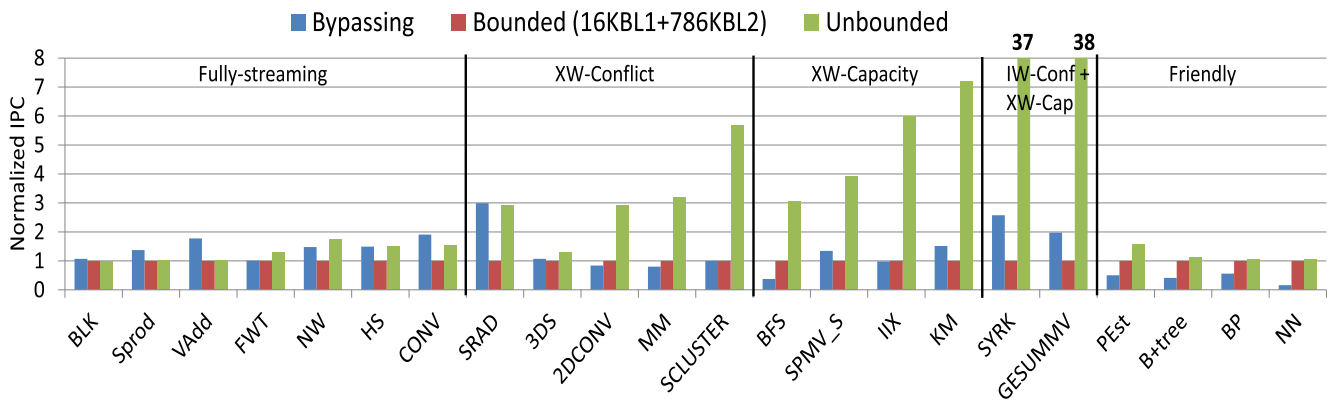


Fig. 2. Studying cache sensitivity for three different scenarios (1) cache bypassing, (2) bounded cache and (3) unbounded cache.

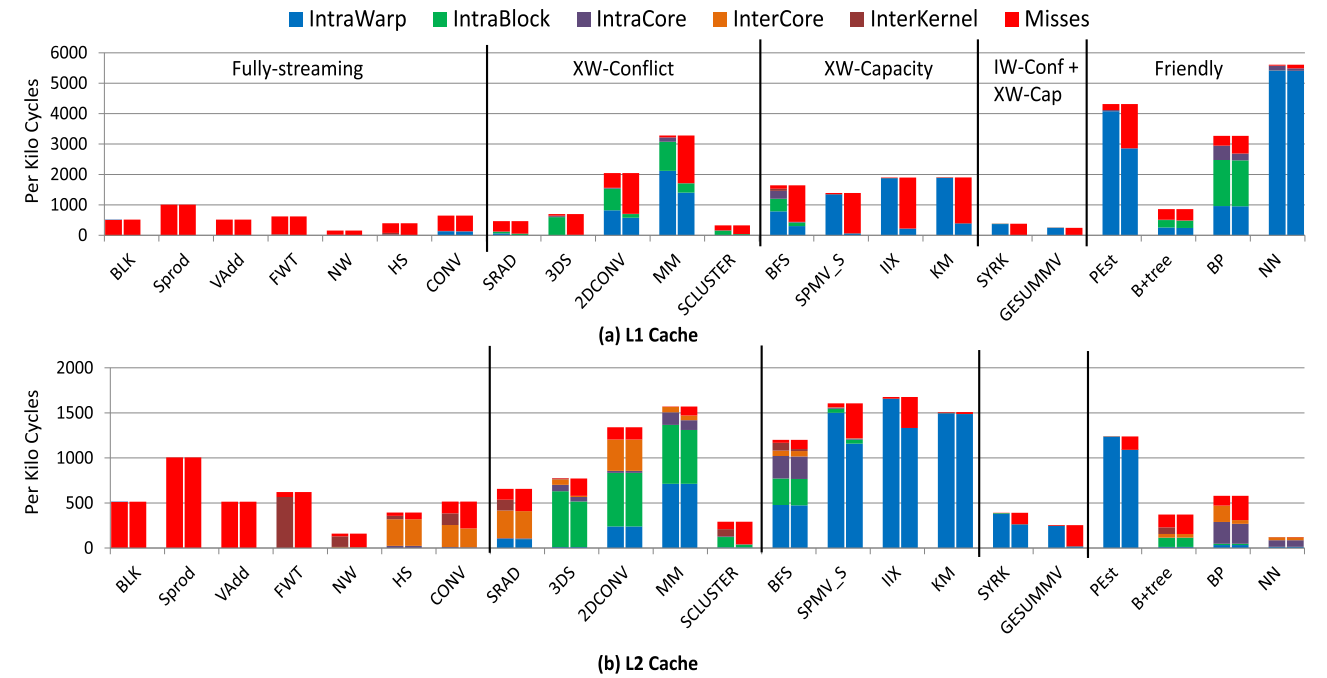


Fig. 3. L1/L2 data locality analysis. The left bar represents the locality found in unbounded caches while right bar for bounded caches.

locality in bounded caches. The L2 cache was evaluated when the L1 cache is bounded.

GPGPU workloads exhibit a high level of contention at the few available L1 cache resources, e.g., MSHR entries, cache lines and Miss_Queue entries. When a cache miss occurs, the MSHRs are checked to see whether the same request has already been issued for another warp and is still pending. If the request is found, a MSHR_MIX entry is allocated to ensure that the returned request services both warps. If the request is not found in the 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 the lack of any requested resource. There are four types of reservation fails. MSHR_MIX_ENTRY_FAIL,MSHR_ENTRY_FAIL, LINE_ALLOC_FAIL, and MISS_QUEUE_FULL occur when the controller fails to allocate MSHR_MIX entry, MSHR entry, cache line, and Miss_Queue entry respectively. In this case, the memory request causes the pipeline to stall, and it will retry over the next cycles until all the requested resources are available. Fig. 4 plots the L1 reservation failures per kilo cycles for the simulated workloads, while Fig. 5 plots the misses per kilo cycles (MPKI).

3.2 Characterization Results

Using the experimental results shown in Figs. 2, 3, 4, and 5, the simulated GPGPU applications can be classified into three main categories:

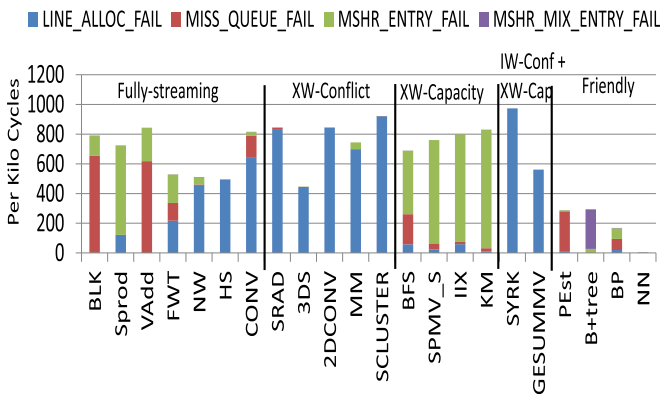


Fig. 4. L1 cache resource contention (using baseline configuration).

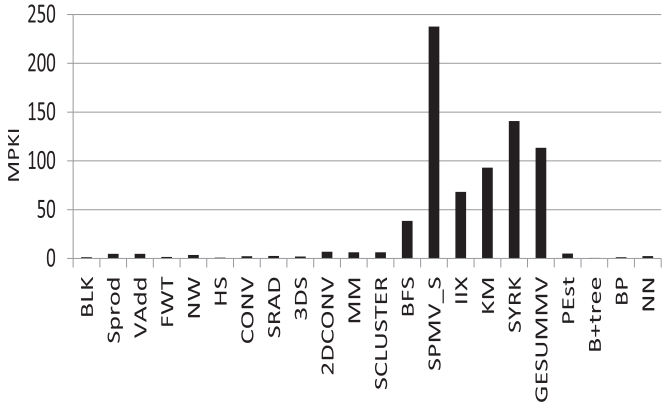


Fig. 5. L1 misses per kilo Instructions (using baseline configuration).

3.2.1 Streaming Applications

Streaming behaviour can occur at L1 and/or L2 caches. There are two types of streaming behavior: fully-streaming and semi-streaming. The former occurs when all the memory access stream generated by the application exhibits an overall high miss rate (up to 99 percent) and does not benefit from caches. The later occurs when a part of the memory requests are full of locality, while the other part shows streaming behavior.

For instance, the BLK, Sprod, and VAdd applications exhibit fully streaming behavior in both L1 and L2. They show a noticeable high miss rate in bounded and unbounded caches, as shown in Fig. 3. These workloads do not benefit from caches at all and using caches degrades the performance. As shown in Fig. 2, using bounded or unbounded caches leads to loss in performance, compared to bypassing. Therefore, it is better to bypass their memory accesses, since they do not benefit from caches and cause useless unintended contention at L1, as shown in Fig. 4.

For FWT and NW, the unbounded cache is better than both the bounded cache and bypassing. These applications exhibit fully streaming behavior in bounded and unbounded L1 cache. However, they have high inter-kernel locality at unbounded L2, whereas bounded L2 is not large enough to cache the data transferred between kernels (see Fig. 3b), causing a fully streaming behavior. In this work, both applications are bypassed. For future work, it is recommended to improve L2 cache behavior by making use of inter-kernel locality.

HS and CONV exhibit fully streaming behavior in L1 (see Fig. 3a), and a high hit rate at L2 (see Fig. 3b). They have high inter-core locality and, thus, they are L2 cache sensitive. Therefore, it is better to bypass L1 cache only for these applications. Inspection of the code of these workloads reveals that they rely on the on-chip scratchpad memory to benefit from locality. Therefore, when they are cached in the scratchpad, they show streaming behavior in L1. It is important to note that the application that uses scratchpad does not necessarily show streaming behavior in L1. For instance, BP relies on scratchpad and exhibits high locality at L1.

Other applications, such as SRAD and BFS, exhibit semi-streaming behavior. In Section 4.3, we will discuss this behavior in more details.

3.2.2 Cache Contention Applications

For cache contention applications, such as SCLUSTER, IIX, and SYRK, the unbounded cache outperforms the bounded cache and cache bypassing by an order of magnitude, as shown in Fig. 2. These workloads have high data locality at the unbounded L1, as shown in Fig. 3. However, the limited size of the bounded L1 cache and the large number of threads a GPGPU executes concurrently makes such workloads susceptible to conflict and capacity misses [29].[3]

Conflict misses mainly occur when a group of warps or a group of threads within the same warp access the same cache set within a short period of time, causing inter-warp conflict contention (*XW Conflict*) or intra-warp conflict contention (*IW Conflict*) respectively [13], [30]. The warps/threads compete for the few available cache lines in the cache set (typically between 4-6 lines [7]). Consequently, a high level of LINE_ALLOC_FAIL stalls (associativity stalls) is observed. Increasing the associativity of the L1 cache is capable of alleviating this type of contention [14].

Capacity misses mainly occur when the cache footprint per-warp is large, leading to inter-warp capacity contention (*XW Capacity*). In this case, no conflict contention occurs but the L1 cache cannot fit all the running warps' working set. Therefore, the warps will compete for the cache lines and the cache becomes susceptible to thrashing. Cache thrashing causes a high level of MSHR_ENTRY_FAIL due to the large number of misses compared to the available MSHR entries. Increasing the L1 cache capacity is capable of alleviating this type of contention [14]. Moreover, it has been shown that the code style is directly linked to either alleviating or increasing cache contention [14]. Writing a highly-divergent non-optimized code, e.g., programs that contain loops, may cause severe cache contention [11].

Based on this conflict analysis, the workloads SRAD, 3DS, 2DCONV, MM, and SCLUSTER suffer from inter-warp conflict contention. These applications exhibit high intra-block locality at unbounded L1, as shown in Fig. 3a. However, the bounded L1 does not benefit from this locality. Fig. 4 demonstrates that these applications exhibit a high level of LINE_ALLOC_FAIL stalls, which indicates the occurence of inter-warp conflict contention.

On the other hand, BFS, SPMV, IIX, and KM suffer from inter-warp capacity contention. They exhibit high intra-warp locality at the unbounded cache (see Fig. 3), which are mostly intra-thread (not shown in figure). The running warps evict the cache lines of each other, and cause severe thrashing at the bounded L1 (see Fig. 3a) and high levels of MSHR_ENTRY_FAIL stalls (see Fig. 4). Fig. 5 indicates that these thrashing workloads exhibit a higher level of MPKI compared to other streaming applications. This is because the streaming workloads are typically optimized to generate fewer well-coalesced memory requests, whereas thrashing workloads generate a large number of non-coalesced memory requests within a few instructions due to memory divergence. That is, a load instruction in streaming applications generate 1-4 missed requests, whereas a load instruction in thrashing applications may generate up to 32 missed requests. Thus, the misses over the committed instructions

3. In this paper, the way we identify conflict and capacity misses is different from the literature [14].

for thrashing applications is relatively higher than streaming workloads. Therefore, MPKI can be used as a good measure to detect thrashing.

Finally, 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. Section 4.2 discusses the behavior of these workloads in details. Note that, in thrashing applications, the L2 cache backs up the 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, as shown in Fig. 3b.

3.2.3 Cache-Friendly Applications

For Pest, B+tree, BP, and NN, the performance of the bounded cache is much better than that of cache bypassing, and is comparable to that of the unbounded cache. These workloads exhibit a high hit rate at both L1 and L2, and their L1 reservation fails are reasonable.

Table 2 summarizes the contention type and streaming behavior of all the GPGPU workloads discussed above.

4 The SACAT Cache Management Scheme

This section describes the three techniques used in the proposed SACAT cache management scheme to bypass streaming behavior, alleviate thrashing and avoid conflicts.

4.1 Dynamic Warp Throttling via Cores Sampling

Warp throttling was proposed as an effective method to alleviate the cache thrashing problem [10], [11]. The number of active running warps per core is throttled to a lower number, such that their cache footprint can be fit in the cache. Static warp throttling (SWT) (a.k.a., Best static warp limiting [10]) 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. On the other hand, dynamic warp throttling (DWT) aims to find the best number of active warps dynamically using the hardware. CCWS is an implementation of DWT [10]. CCWS uses a victim tag array, called the lost locality detector, to detect warps that have lost locality due to thrashing. These warps are prioritized until they exploit their locality, while other warps are descheduled (not allowed to issue any load instructions).

CCWS has a fine-grained control on warp throttling (i.e., the number of active warps is varied over time depending on the thrashing level). This gives CCWS an advantage over SWT. However, it has been shown that the coarse-grained SWT outperforms CCWS on average [10]. This is due to the fact SWT tries to find the best trade-off number of warps that works well at different thrashing levels and improves the overall performance. Moreover, CCWS is a reactive system, i.e., each warp has to lose locality before trying to preserve it by warp throttling [11], whereas SWT throttles the number of warps directly with no need for lost locality detection. However, SWT is not a practical solution, since the programmer needs to perform 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 [10].

TABLE 3
The Number of Active Warps (per Warp Scheduler) Achieved by SWT, DWT-CS and CCWS

Benchmark	SWT	DWT-CS	CCWS
SPMV	1	1	1-4
Kmeans	1	1	1-2
BFS	5	7	2-15
IIX	2	2	1-7
SYRK	2	2	2-4
GESUMMV	1	1	1-2

In this work, we propose dynamic warp scheduling via core sampling technique that employs exhaustive search as in SWT. However, the search process is handled by the hardware. Therefore, DWT-CS overcomes the shortcomings of SWT. The concept of cores sampling was proposed by Lee, et al., [31]. It applies different cache management policies to different cores, and collects samples to assess their behavior. In this work, a similar mechanism is employed to find the best number of active warps that alleviates thrashing and efficiently utilizes the L1 cache.

4.1.1 DWT-CS Mechanism

As shown in Fig. 5, L1 MPKI can be used as a good measure to detect thrashing. Therefore, DWT-CS monitors the MPKI at L1 over several sampling periods. 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 numbers of active warps, which is equal to the core ID. For example, core#1 throttles the active warps to only one warp, core#2 throttles them to two warps, and so on. After M sampling periods, all cores send the number of instructions committed during the sampling periods to the coordinator core, which is 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, which is used until the end of kernel execution. When the same kernel is relaunched and the MPKI exceeds the threshold, it does not sample the cores again. Instead, it uses the same number of warps obtained before.

4.1.2 Comparison with SWT and CCWS

Table 3 compares the best number of active warps achieved using SWT, DWT-CS as well as the range of active warps under CCWS. DWT-CS achieves 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). For BFS and other non-steady thrashing applications, changing the number of active warps over time based on the thrashing level, like CCWS does, is slightly better than fixing the number of warps as in DWT-CS. In contrast, for steady thrashing applications, such as Kmeans and IIX, applying constant number of active warps, as in DWT-CS, is more effective than varying the number of warps. This is because DWT-CS does not impose overhead to detect

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

Fig. 6. Memory locations interleaving over cache sets (Assuming a 6-bit memory address, a 1-byte cache line, and eight cache sets.)

thrashing all the time and adjust the number of active warps. Instead, it detects thrashing only once at the beginning of kernel execution. We will compare the performance of CCWS against DWT-CS in Section 5.2.

4.1.3 Implementation Overhead

The proposed DWT-CS technique is a cost-effective method. It slightly outperforms CCWS on average over thrashing applications, as will be discussed in Section 5, while requiring negligible hardware overhead. More specifically, CCWS uses extra hardware (victim tag arrays) to detect thrashing, whereas DWT-CS only needs two counters per core to calculate the committed instructions and cache misses, in order to measure the MPKI over sampling periods. A few registers are also used to save the best number of warps per-kernel for future reuse. To enable the inter-core communication that is used to convey sampling results, we rely on the existing interconnection between the Block Scheduler and the cores (see Fig. 1). That is, all the cores send the IPC to the Block Scheduler (a centralized node), then the Block Scheduler sends these information to the nearest core (assuming the middle core). Afterwards, the middle core uses its SIMD unit [18] to find the best core that achieved the highest IPC. We modeled this inter-core communication with a constant number of n cycles. In our experiments, we set n=200 cycles. DWT-CS requires about 60 K cycles to detect thrashing and find the optimal number of warps, which is a negligible overhead in practice.

4.2 Pseudo Random Interleaved Cache

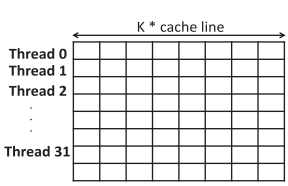
The problem of CPU cache associativity has been widely studied in the literature. Several techniques were proposed to improve the cache indexer function, which is responsible for interleaving memory accesses over cache sets. These techniques include prime modulo interleaving [32], one-skew storage [33], logical data skewing [34], Xor-based functions [35] and pseudo random interleaving [17]. They were proposed as alternatives to conventional sequential interleaving, in an attempt to improve cache associativity and avoid conflicts. It has been shown that PRIC is a cost-effective high-performance approach [177], [36], [37]. In this work, we study the impact of applying PRIC for GPU caches and see the effectiveness of PRIC in alleviating associativity stalls and eliminating conflict misses.

4.2.1 PRIC Mechanism

In a sequential interleaving cache consisting of $M = 2^m$ cache sets and a cache line size of $B = 2^b$ bytes, an N-bit

```
//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

Fig. 7. An example of a one-way conflict degree in the SYRK workload. When K is multiple of the number of cache sets, all 32 threads will map to the same cache set.

memory location whose address is A[$N-1$:0], has a cache index A[$m+b-1$:b] and a tag address A[$N-1$:$m+b$]. Fig. 6a depicts a simple example of how memory locations are interleaved over cache sets using sequential interleaving. An application that generates a stream of M memory references in a short period of time with an access stride S, has an n-way conflict degree, where n=$M/\gcd(M,S)$, and gcd stands for the greatest common divisor. From this relation, it can be easily observed that even strides will cause a high level of conflict degree. For example, using the sequential memory interleaving shown in Fig. 6a, a reference stream with an access stride of 2 (i.e., 0, 2, 4, 6, 8, 10, 12, 14) has a four-way conflict degree. In other words, all memory references will be mapped to only four 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, if the cache set contains fewer cache lines than the mapped memory references. The worst-case scenario occurs when the reference sequence has a stride which is a multiple of M. This causes a one-way conflict, where all references map to the same cache set. On the other hand, all odd strides have no common divisor with M that is greater than one (recall that M is a power-of-2 number). Therefore, odd strides do not 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 multiples of M, occur frequently in GPGPU applications. For instance, Fig. 7 shows a GPGPU frequently occuring 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 rows of A are aligned to the cache line size (i.e., row size=$K\times$line size) and are stored in a row-major order form in the main memory. In 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 the baseline configuration), all 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 $32\times n\times$line size, where $n >=1$, frequently exists in GPGPU applications.

In PRIC, as shown in Fig. 6b, each M consecutive memory locations have a different permutation over the cache sets such that they are near-randomly interleaved. This near-random interleaving makes PRIC resistant to all strides, especially strides that are multiples 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

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

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

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

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

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

Fig. 8. The Xor-ing equations corresponding to Poly(37).

with memory location A. Then, $A(x)$ can be uniquely represented as $A(x) = V(x) \times P(x) + R(x)$. 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 viewed as the polynomial representations of the corresponding tag and cache index. Therefore, the cache index of address $R(x) = A(x) mod P(x)$. It has been shown 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 $P(x) = g(x) \times h(x)$ exist [38]. Rau [17] showed how the computation of cache index $R(x) = A(x) \mathrm{mod} P(x)$ can be carried out by the vector-matrix product of the address and a matrix of single-bit coefficients, called the H-matrix (i.e., I[$m-1$:0] = A[$N-1$:b]$\times$ H-matrix). In GF(2), multiplication and addition are equivalent to the AND and XOR boolean functions. 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 [36].

4.2.2 Implementation Overhead

The baseline configuration has $32 = 2^5$ cache sets, thus, m=5. There are six irreducible polynomial functions of degree 5 over GF(2) [38] and they are Poly (37, 41, 47, 55, 59, 61). In this work, Poly(37) is selected empirically. The corresponding Xor-ing boolean equations of Poly(37) are listed in Fig. 8.[4] As shown in the figure, the cache index I[4:0] is generated by Xor-ing some bits of the memory address A[26:7]. These Xor-ing equations can be implemented using 2-3 levels of two-input Xor gates.

It is important to note that one of the limitations to employing PRIC in a single-thread CPU cache is that it exists on the critical path of any cache access. Therefore, any cache access latency may increase by one cycle due to the latency of the Xor gates [36], [37]. This latency can degrade the performance of some non-conflict cache-friendly applications (i.e., applications that don't benefit form PRIC). However, when it comes to GPGPUs, the situation is different. GPGPUs have a throughput-oriented architecture. Their design philosophy is built on having a large number of warps/threads per core that are interleaved with each other in order to hide long memory latency. We experimentally study the performace impact when we increase the cache access latency by one cycle. Only 0.05 percent performance loss was observed for friendly applications.

4.2.3 Comparison with Cache Bypassing

Cache bypassing on associativity stalls was proposed recently [14]. However, this method does not utilize cache resources efficiently. In many cases, bypassing occurs while the other cache sets are underutilized. For instance, MRPB [14] allows memory requests that encounter associativity stalls (i.e., LINE_ALLOC_FAIL) to bypass the L1 cache [14]. In the SYRK workload, shown in Fig. 7a, when all the 32 threads map to the same cache set, only the first four threads will successfully allocate a cache line (assuming four-way associativity), while the remaining 28 threads will bypass the L1 cache. This occurs because all the lines within the cache set will be reserved by the first four threads. However, performing an empirical search reveals that the other cache sets are underutilized. Therefore, it is better to distribute the remaining threads over the underutilized cache sets, instead of bypassing. Moreover, in the second iteration, the same 32 threads access the following 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 misses, and consequently, they evict the previous threads' cache lines. This behavior is repeated over the next loop iterations, i.e., 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 Fig. 3a. Hence, cache bypassing is not an efficient method to handle the GPU cache associativity problem.

4. The proofs and theorems related to the polynomial modulus and the method to generate the H-matrix and Xor-ing equations can be found in [17].

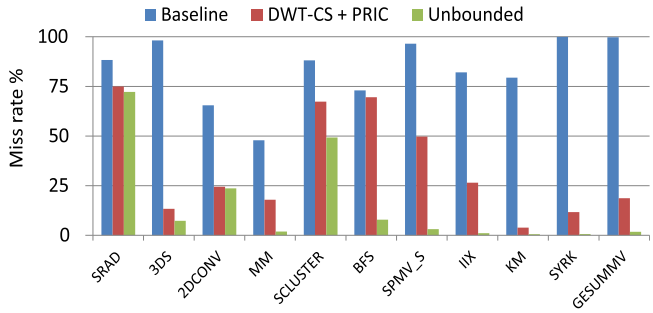


Fig. 9. The L1 miss rate after applying DWT-CS and PRIC.

4.2.4 Comparison with High Associativity Caches

Increasing the associativity of the L1 cache is a straightforward approach to mitigate associativity stalls and conflict misses. Moreover, building a fully associative cache can completely eliminate all conflict misses. However, increasing associativity requires a considerable hardware overhead (tag comparators and large data selectors), which increases both access latency and power consumption [39]. On the other hand, PRIC can achieve almost the same performance of fully associative cache (as will be discussed in Section 5.2), while incurring low hardware overhead (2-3 levels of Xor gates).

4.3 Dynamic Fine-Grained Cache Bypassing (FG-CB)

Fig. 9 plots the miss rate of the L1 cache after applying DWT-CS and PRIC, compared to the baseline case and the unbounded cache. The figure shows that some applications benefit from the proposed throttling-resistant and conflict-avoiding techniques. 3DS, KM, SYRK, and GESUMMV achieve a low miss rate that is comparable to that of the unbounded cache. On the other hand, other applications still exhibit a high miss rate with DWT-CS and PRIC. For example, SRAD, SCLUSTER, BFS, and SPMV exhibit reduction in the miss rate using DWT-CS and PRIC, compared to

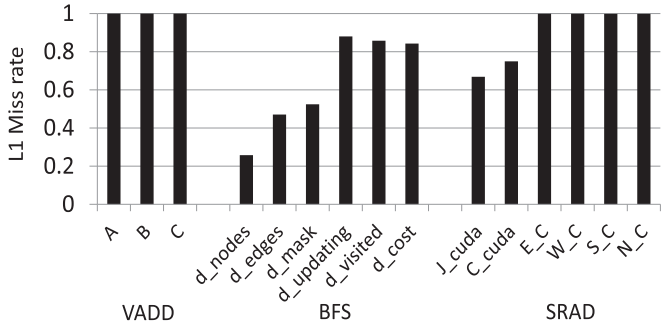


Fig. 10. The L1 miss rate per-base-address for fully- and semi- streaming workloads.

the baseline. However, this reduced miss rate is still high, i.e., 75, 70, 68, and 50 percent respectively. There are two reasons behind these high miss rates. First, SRAD and SCLUSTER suffer from noticeable streaming behavior (up to 70 and 50 percent miss rate in the unbounded cache, respectively). It is better for an application, such as SRAD, to bypass the 70 percent that shows streaming behavior and cache the the remaining 30 percent that has good locality. Second, some applications have high locality which cannot be fully utilized by DWT-CS and PRIC. For example, BFS shows a miss rate of 5 percent in the unbounded cache. However, a large portion of the locality found in this application is not utilized by the cache due to the random behavior of some memory accesses [14], and the long reuse interval between memory requests that access the same cache line. The proposed workload characterization methodology reveals that these streaming memory accesses do not only cause useless contention at L1/L2 caches, but also severely interfere with cache-friendly memory accesses. Therefore, the SACAT scheme employs a dynamic fine-grained cache bypassing technique to alleviate streaming behavior at fine-granularity.

4.3.1 Fine-Grained Cache Bypassing Mechanism

The straightforward approach to dynamically bypass streaming behavior is to enable or disable the whole cache. In this method, when the overall miss rate of cache is higher than a predefined threshold, the whole cache is disabled. The coarse granularity of bypassing in this method can effectively deal with fully-streaming applications. However, it is not efficient with semi-streaming workloads that require a finer granularity to bypass only the streaming part of memory requests and cache the other part containing locality.

Using FG-CB, a memory request is bypassed based on the miss rate of the base-address that this request belongs to. For example, a kernel takes three pointers of arrays KernelF$<<<$(n,1),(m,1)$>>>$(float$*$ A, float$*$ B, float$*$ C) has three base-addresses (A, B and C). Any load instruction in kernelF should be referred to one of these addresses as a base address. Fig. 10 plots the miss rate for the fully-streaming workload VAdd, and two semi-streaming workloads, BFS and SRAD. As shown in figure, VADD shows a high miss rate over all base-addresses (A, B and C). Coarse-grained or find-grained cache bypassing can efficiently bypass all memory requests of VADD. On the other hand, SRAD has a noticeable streaming behavior at some base-addresses (E_C, W_C, S_C and N_C), whereas J_cuda and C_cuda contain some locality that can be utilized by caches. A similar behavior also exists in BFS. Therefore, A fine-grained cache bypassing mechanism is required for such applications.

In the SACAT implementation, instead of monitoring the miss rate of the whole cache, the miss rate of every base-address is calculated. When a new kernel is launched, the miss rate of every base-address found in the kernel is sampled. Based on the experiments performed, it was found that the miss rate calculated in the first sampling period can by used as a good measure to anticipate the streaming behavior of the base-address during the rest of the execution time. To confidently determine whether to bypass a base-address or not, at least *N_Accesses* of memory requests generated to this base-address should be observed. If the number of misses of any base-address is greater than *M_Misses*, at the first sampling period, all memory requests belonging to this base-address are forced to bypass the cache until the end of the kernel. Otherwise, the requests are cached. The values of *N_Accesses* and *M_Misses* used in this experiment were 1,000 and 800, respectively.

4.3.2 Implementation Overhead

In FG-CB, a hardware-based table is used to monitor the miss rate of the L1 and L2 caches for the first sampling period. Each table entry corresponds to one base-address and contains five fields: the base-address value, the number of cache accesses at L1/L2, and the number of cache misses at L1/L2. The number of table entries can be fixed. The workloads considered here have at most seven base-addresses.

4.3.3 Per-Base-Address versus Per-PC Cache Bypassing

Dynamic cache bypassing at Per-PC load instruction granularity was proposed in [40]. However, our experiments indicate that the per-PC cache bypassing can be ineffective for some workloads, e.g., BFS and SRAD. In such applications, a dependency exists between a streaming PC-load instruction and a cache-friendly PC-load instruction. In other words, the cache line that is brought by the streaming PC-load instruction is re-referenced by a consecutive cache-friendly PC-load instruction that has the same base-address. In this scenario, if the first PC-load is forced to bypass due to its streaming behavior, the consecutive PC-load will not able to utilize its locality. This scenario is avoided in the per-base-address mechanism, since the entire base-address region is either cached or bypassed.

4.4 Putting it All Together (SACAT)

The FG-CB, DWT-CS, and PRIC techniques are combined in one algorithm, called SACAT. This combines the advantages of the individual techniques and achieves the highest performance for the GPGPU workloads. It is important to note that both streaming and thrashing applications exhibit a high miss rate at L1 (as shown in Fig. 3). Therefore, FG-CB and DWT-CS must be combined appropriately to avoid the mischaracterization of a thrashing application as a streaming one. First, the conventional sequential cache indexer is replaced by PRIC, such that each cache access has to pass through the PRIC xoring equations to avoid conflict misses and stalls. Second, an application is examined to determine whether it is thrashing or not. To achieve this, SACAT

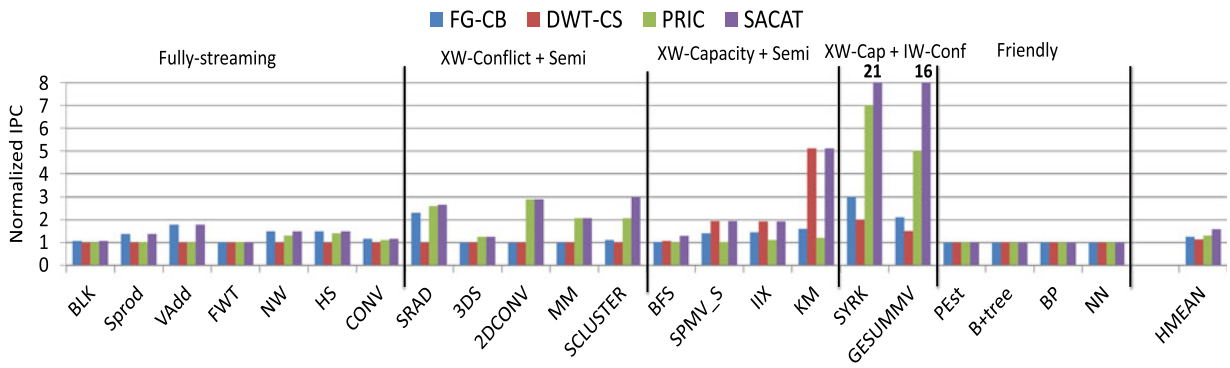


Fig. 11. The performance improvement achieved using each technique individually.

monitors the MPKI of L1 for the first N sampling period. If it exceeds a specific threshold for all N periods, the application is characterized as thrashing, and DWT-CS runs an exhaustive search to find the best number of warps. This approach leverages the fact that thrashing applications exhibit a high MPKI over other streaming applications (as shown in Fig. 5). After DWT-CS runs for the first N sampling periods to find the appropriate number of warps that alleviates thrashing, if any, FG-CB runs for the following sampling period. It monitors the miss rate of every base address at the L1 and L2 caches using the hardware-based per-base-address table. If any of the base addresses achieves *N_Accesses* memory requests, the number of misses that occurred during this interval is calculated. If that number exceeds *M_Misses*, the cache controller is updated to bypass all the memory requests belonging to this base-address until the end of the kernel. Otherwise, it is cached.

The three proposed techniques are grouped together in an orthogonal and synergistic way. SACAT does not only cache a part of the memory that contains locality but also caches them in an efficient manner in order to avoid thrashing and conflict.

5 Experimental Results

The experimental results are organized as follows. Section 5.1 examines the performance of the proposed SACAT cache management scheme compared to the baseline and presents a detailed analysis of its advantages. Section 5.2 compares the proposed SACAT technique to previous work. Section 5.3 studies the sensitivity of L1 cache size. Finally, Section 5.4 is devoted to calculating total hardware overhead.

5.1 In-Depth Analysis

Fig. 11 presents the performance improvement (in IPC with respect to the baseline) of the proposed techniques, FG-CB, DWT-CS, PRIC, and the aggregated SACAT, for all benchmarks. FG-CB, DWT-CS and PRIC improve the performance of GPGPU applications by 1.25×, 1.13× and 1.3×, respectively, over the baseline. When all these techniques are aggregated in the SACAT algorithm, to combine their advantages, the performance improvement is about 1.6× on average. The applications SYRK and GESUMMV exhibit an improvement of up to 21X and 16.8X, respectively. Moreover, SACAT does not result in any performance degradation in the cache-friendly applications.

An individual technique may be sufficient for some applications, while other applications require a combination of two or more techniques. Fully-streaming applications only benefit from FG-CB. Since the L1 cache is disabled in FG-CB, such applications do not suffer from thrashing nor conflicts. PRIC and FG-CB show performance improvement for inter-warp conflict workloads, as opposed to DWT-CS. However, PRIC provides better performance improvement than FG-CB on average. Inter-warp capacity conflict applications (i.e., thrashing-only applications) show a significant performance improvement using DWT-CS only, compared to a slight improvement using FG-CB. Applications that exhibit both intra-warp conflict contention and inter-warp thrashing (e.g., SYRK and GESUMMV) show a higher performance improvement when employing FG-CB and PRIC than that achieved using DWT-CS. Moreover, they show a superior performance improvement using SACAT. In these workloads, PRIC eliminates conflict misses and fairly distributed memory access over cache sets. Nevertheless, the L1 cache capacity is not large enough to keep all warps' working set and, thus, DWT-CS throttles the number of active warps to alleviate thrashing. Therefore, these workloads achieve a significant performance improvement when both DWT-CS and PRIC are combined. In other words, neither DWT-CS nor PRIC alone is able to individually achieve the maximum performance improvement for these applications.

Fig. 12 plots the reduction in L1 reservation fails (all fails shown in Fig. 4 combined) when combining DWT-CS and PRIC, as well as when applying SACAT. Combining DWT-CS and PRIC reduces the reservation fails by 80 percent on average. By adding FG-CB, i.e., using SACAT, reduces the reservation fails further to 90 percent. Applications such as 3DS, 2DCONV, IIX, KM, SYRK and GESUMMV show a significant reduction (up to 90 percent). SRAD, MM, SCLUSTER, BFS and SPMV_S also show a considerable reduction, between 20 and 60 percent, when combining DWT-CS and

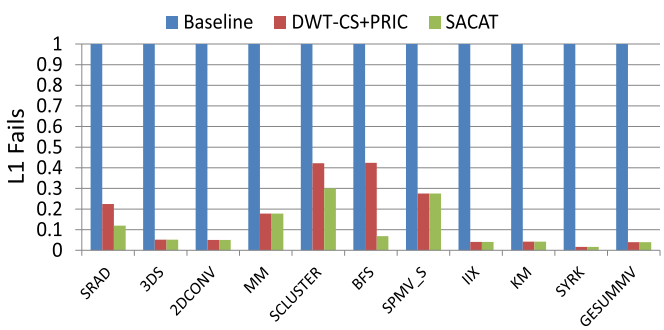


Fig. 12. The L1 reservation fails (normalized to the baseline).

TABLE 4
The Configuration Parameters of CCWS, MRPB, and SACAT

CCWS Config	
Kthrottle	8
Victim Tag Array	8-way 16 entries per warp (768 total entries)
Warp Base Score	100
Cache indexing	sequential interleaving
MRPB Config	
Signature	warp ID (resulting in 48 queues)
Drain policy	non-greedy-fixed-order
Buffer size	8 requests
Bypass option	bypass-on-assoc-stalls
SACAT Config	
DWT-CS Sampling Period	10 K cycles
DWT-CS MPKI_threshold	10
FG-CB N_Accesses	1000
FG-CB N_Misses	800
PRIC I-Poly	Poly(37)

PRIC. Moreover, SRAD, SCLUSTER and BFS exhibit further reduction using SACAT, due to the efficiency of FG-CB in bypassing streaming memory accesses that cause useless contention at the L1 cache. However, MM, SCLUSTER and SPMV_S still show a high level of fails, especially MSHR_ENTRY_FAIL. These applications still suffer from some sort of thrashing and streaming contention even after applying SACAT. Further investigations and applying more advanced techniques can mitigate the remaining MSHR contention.

5.2 Comparison to Previous Work

The proposed SACAT technique was compared to CCWS [10] and MRPB [14]. As discussed in Section 4.1, CCWS addresses the thrashing problem only and does not consider conflict contention nor streaming behavior. 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 such that requests from the same warp are grouped and sent to the cache together, thereby reducing the number of warps that access the cache at a time. Second, MRPB allows memory requests that encounter associativity stall to bypass the L1 cache. However, the bypassing strategy does not efficiently utilize the available cache resources, as discussed in Section 4.2. Table 4 lists the configuration parameters used for CCWS, MRPB, and SACAT. In CCWS, the value Kthrottle was tuned to our baseline architecture as described in [10], and sequential set indexing is used. In MRPB, the configuration in [14] is used, and it achieved the highest performance. In SACAT, the configuration parameters were selected based on empirical analysis. To ensure fair comparison, fully-streaming applications were excluded, since CCWS and MRPB do not address the streaming behavior problem.

Fig. 13 compares the performance improvement achieved in cache contention applications using DWT-CS and SACAT to that of CCWS and MRPB. Overall, SACAT outperforms CCWS and MRPB by a harmonic mean of 1.87× and 1.5×, respectively. For inter-warp conflict contention applications, SACAT improves performance by a harmonic mean of 2.3×

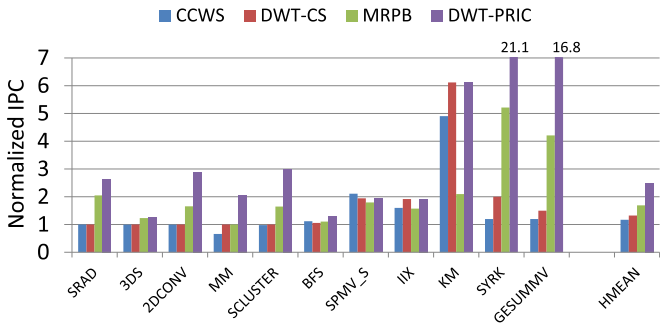


Fig. 13. Performance improvement (normalized to baseline) of SACAT compared to CCWS, MRPB, and DWT-CS.

and 1.4× over CCWS and MRPB, respectively. This is due to CCWS's unawareness of conflict contention, and the better efficiency of PRIC in utilizing cache sets over MRPB's cache bypassing mechanism. For inter-warp thrashing applications, SACAT results in a harmonic mean 1.11× and 1.35× performance improvement over CCWS and MRPB, respectively. When we compare CCWS to DWT-CS only, it exhibits 1.05× performance improvement on average. DWT-CS shows a significant improvement in applications that exhibit a consistent thrashing level and coherent control flow divergence, e.g., IIX and KM. On the other hand, CCWS performs slightly better in SPMV and BFS, due to the unsteady thrashing level of these applications. SPMV and BFS are highly control flow divergent and thus the per-warp cache footprint changes over time depending on warp's active mask. DAWS is a divergence aware warp throttling mechanism that can further improve the performance of such applications [11]. Note that, in BFS, SACAT slightly outperforms CCWS and DWT-CS because FG-CB bypasses streaming memory accesses efficiently. For applications that exhibit both intra-warp conflict contention and inter-warp thrashing, SACAT achieves a superior performance improvement and outperforms CCWS and MRPB by a harmonic mean 18.7× and 4×, respectively.

Increasing the associativity of the L1 cache is a straightforward approach to mitigate conflict misses. For example, AMD's recent Graphics Core Next (GCN) GPUs use 64-way associativity for their 16 KB L1 caches [41]. Fig. 14 compares the performance improvement obtained using PRIC on inter-warp and intra-warp conflict contention applications, to that obtained using high associativity and fully-associative caches. In all cases, the L1 cache capacity is fixed and an idealistic one-cycle hit latency is assumed. PRIC with four-way associativity outperforms the 16-way, 32-way, and 64-way caches by

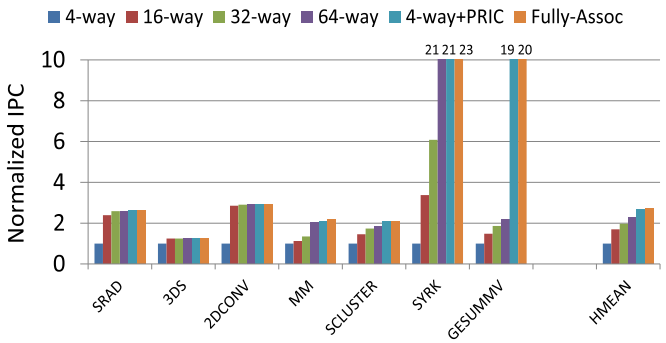


Fig. 14. Comparing the performance of applying PRIC to high and full associativity (normalized to baseline).

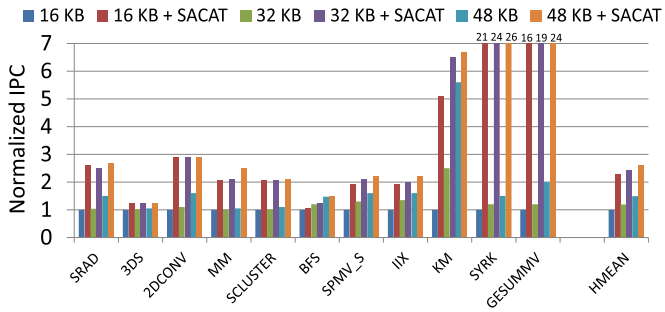


Fig. 15. L1 cache sensitivity (normalized to baseline). The cache configuration of 16, 32, and 48 KB caches are 32sets-4way, 64sets-4way and 64sets-6way respectively.

a harmonic mean 1.6×, 1.4×, and 1.16×, respectively. Moreover, PRIC achieves 97 percent of fully-associative cache's performance. However, increasing associativity requires a considerable hardware overhead (tag comparators and large data selectors) which increases both access latency and power consumption [39].

5.3 Sensitivity to Cache Size

Fig. 15 depicts the performance of baseline and SACAT at various cache sizes. The results are normalized to the baseline architecture with a 16 KB L1 cache. The performance improvements of SACAT over the baseline with different cache size are 2.3×, 2× and 1.7× for 16, 32 and 48 KB respectively. Consequently, as the cache size decreases, SACAT shows greater performance improvement relative to the baseline architecture. This is because smaller L1 cache size increases the occurrence of cache thrashing and conflicts.

5.4 Overall Hardware Overhead

Table 5 shows the total hardware overhead of SACAT scheme. Overall, SACAT requires 5,270 bits of SRAM bits (i.e., less than 1 KB). In addition, it requires some combinational logic circuitry to implement the PRIC's Xor gates and other control circuits, and we do not expect this additional logic will be significant. On the other hand, CCWS requires 3.75 KB per-core of SRAM bits to implement Victim Tag array (768 entries × 40 bits per entry [11] = 3.75 KB), and MRPB requires 1.875 KB per-core of SRAM bits to implement FIFO queues (48 queues × 8 requests per queue × 40 bits request size [14]). Thus, SACAT is a cost-effective method.

TABLE 5
Overall Hardware Overhead

Technique	Hardware Overhead	Total (#bits)
DWT-CS	MPKI calculation: 2 counters (per-core) × 16 bits × 15 cores = 480 bits Table to save the best #number of warps per-kernel for future reuse: 8 bits (to save kernel Id) + 5 bits (#number of warps) = 13 bits × 5 entries = 65 bits	545 bits
PRIC	no registers required, only 2-3 levels of Xor gates per core	0
FG-CB	Miss rate per-base-address table: 5 bits (base-address Id) + 10 bits (L1 cache accesses) + 10 bits (L1 cache misses) + 10 bits (L2 cache accesses) + 10 bits (L2 cache misses) = 45 bits × 7 entries = 315 bits × 15 cores = 4,752 bits	4,752 bits

6 Related Work

Different methods have been proposed in the literature to alleviate the problems associated with GPU caches. These methods can be classified into the following four categories:

6.1 CTA Throttling

Jog, et al., proposed CTA-aware-locality scheduling [42]. It 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., proposed dynamic CTA scheduling, which attempts to allocate the optimal number of CTAs per-core in order to reduce contention in the memory sub-system [8]. Lee, et al., explored two alternative thread block scheduling schemes [9]. Lazy CTA scheduling was proposed to leverage the GTO scheduler to determine the optimal number of CTAs per core. They also showed how block CTA scheduling (BCS), where consecutive thread blocks are assigned to the same cores, can utilize inter-block locality (i.e., intra-core and inter-core locality). It is obvious that the fine-grained warp throttling mechanisms, such as DWT-CS, are better than the coarse-grained CTA throttling mechanisms. Based on experiments (not shown here), static warp throttling was found to outperform static CTA throttling by 2× on average.

6.2 Warp Throttling

In addition to the CCWS technique discussed above, several other techniques were proposed to improve warp throttling. Rogers, et al., proposed divergence-aware warp scheduling (DAWS) [11]. DAWS is a divergence-based cache footprint predictor that estimates the amount of locality in loops required by each warp. DAWS uses these predictions to prioritize a group of warps such that the cache footprint of these warps does not exceed the capacity of the L1 cache. Li, et al., observed that throttling techniques leave memory bandwidth and other chip resources (e.g., L2 cache, NOC and EUs) significantly underutilized [12]. Thus, he proposed a cache bypassing scheme on top of CCWS, called priority-based cache allocation (PCAL), that allows extra inactive warps to bypass the cache and utilize the other on-chip resources. Therefore, 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., called adaptive cache and concurrency (CCA) [13]. CCA improves DAWS by allowing extra inactive warps and some streaming memory instructions from the active warps to bypass the L1 cache and utilize the on-chip resources. However, PCAL and CCA employ bypassing while leaving cache sets underutilized. For example, recall the SYRK example discussed in Section 4.2. 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 discussed above, the cached warp only utilizes one cache set. Moreover, it utilizes that cache set in an inefficient manner, since the threads map to the same set causing severe associativity stalls and conflict misses. In contrast, SACAT effectively utilizes the cache sets

by allowing two warps to access the cache, and fairly distributing their memory requests over sets. Note that, PCAL or CCA can be employed on top of SACAT for further performance improvement and efficient utilization of L1 cache sets, as well as on-chip resources.

6.3 FIFO Buffers

The MRPB technique, discussed in Section 4.2, uses FIFO buffers to prioritize memory requests that are generated by the same warp. It also uses cache bypassing on associativity stalls, and it has been demonstrated that the bypassing mechanism is inefficient to utilize the available resources.

6.4 Cache Replacement Policy

Chen, et al., [16] proposed G-Cache to alleviate cache thrashing by identifying the hot lines that have been evicted before and an adaptive cache replacement policy is used by the L1 cache to protect these hot lines. Chen also proposed coordinated bypassing and warp throttling (CBWT) [15]. CBWT adopts a thrashing-resistant CPU cache management scheme, called protection distance prediction (PDP). PDP employs cache bypassing to enable protection on hot cache lines and, thus, alleviates cache thrashing. However, excessive bypassing may over-saturate the on-chip network. Therefore, the cache bypassing policy is coordinated with a dynamic warp throttling mechanism to avoid over-saturating on-chip resources. However, existing schemes do not address the associativity problem, and employ a cache replacement policy to alleviate thrashing, as opposed to SACAT which uses a warp throttling mechanism. Chao, et al., proposed a locality-driven dynamic bypassing solution that augments the L1 tag store with locality filtering. The new hardware component forces the L1 cache to only store the data with high reuse and short reuse distances while bypasses the other requests [43].

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 a GPGPU executes concurrently leads to severe cache thrashing and conflict misses. This paper proposed a low-cost streaming-aware conflict-avoiding thrashing-resistant GPGPU cache management scheme. The proposed scheme efficiently utilizes the GPGPU cache resources and addresses all the problems associated with GPGPU caches, by employing three orthogonal techniques. Experimental results demonstrate that the proposed scheme achieves 1.87× and 1.5× performance improvement over CCWS and MRPB, respectively. Moving forward, we plan to study the impact of running multiple kernels on the GPU cache hierarchy and exploiting the inter-kernel locality.

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 the GPGPU-sim tool. Our original work on GPGPU caches appeared in "Efficient Utilization of GPGPU Cache Hierarchy", at the 8th Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2015).

References

[1] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep.–Oct. 2011.
[2] D. Kirk and W. Wen-Mei, *Programming Massively Parallel Processors: A Hands-On Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2010.
[3] P. Micikevicius, "GPU performance analysis and optimization," in *Proc. GPU Technol. Conf.*, 2012.
[4] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, Mar.–Apr. 2011.
[5] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, NVIDIA Corporation, Santa Clara, CA, USA, 2013.
[6] L. Gwennap, "Sandy bridge spans generations," *Microprocessor Rep.*, vol. 9, no. 27, pp. 10–01, 2010.
[7] R. Meltzer, C. Zeng, and C. Cecka, "Micro-benchmarking the C2070," in *Proc. GPU Technol. Conf.*, 2013.
[8] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 157–166.
[9] 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 *Proc. IEEE 20th Int. Symp. High Performance Comput. Archit.*, 2014, pp. 260–271.
[10] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2012, pp. 72–83.
[11] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 99–110.
[12] D. Li, et al., "Priority-based cache allocation in throughput processors," in *Proc. IEEE 21st Int. Symp. High Performance Comput. Archit.*, 2015, pp. 89–100.
[13] Z. Zheng, Z. Wang, and M. Lipasti, "Adaptive cache and concurrency allocation on GPGPUs," *IEEE*, vol. 14, no. 2, pp. 90–93, 2015.
[14] W. Jia, K. A. Shaw, and M. A. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *Proc. IEEE 20th Int. Symp. High Performance Comput. Archit.*, 2014, pp. 272–283.
[15] 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 *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 343–355.
[16] 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 *Proc. Int. Workshop Manycore Embedded Syst.*, 2014, Art. no. 1.
[17] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proc. 18th Annu. Int. Symp. Comput. Archit.*, 1991, pp. 74–83.
[18] *CUDA C Programming Guide v5.5*, NVIDIA, Santa Clara, CA, USA, 2013.
[19] *The OpenCL Specification Version 2.0*, OpenCL, Beaverton, OR, USA, 2015.
[20] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Performance Anal. Syst. Softw.*, 2009, pp. 163–174.
[21] W. W. Fung, I. Sham, G. Yuan, and W. W. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," *MICRO.*, 2007.
[22] *GPGPU-sim 3. x Manual*, 2012.
[23] S. Che, et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2010, pp. 44–54.
[24] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. Innovative Parallel Comput.*, 2012, pp. 1–10.
[25] NVIDIA, "CUDA C/C++ SDK code samples," (2013). [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples
[26] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 260–269.

[27] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, C. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd Workshop General Purpose Process. Using GPUs*, 2010, pp. 63–74.
[28] D. Li and T. Aamodt, "Inter-core locality aware memory scheduling," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 25–28, Jan.–Jun. 2016.
[29] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.
[30] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 15–24.
[31] J. Lee and H. Kim, "Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture," in *IEEE 18th Int. Symp. High Performance Comput. Archit. (HPCA)*, 2012, pp. 1–12.
[32] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Proc. 10th Int. Symp. High Performance Comput. Archit.*, 2004, Art. no. 288.
[33] D. T. Harper and J. R. Jump, "Vector access performance in parallel memories using a skewed storage scheme," *IEEE Trans. Comput.*, vol. C-36, no. 12, pp. 1440–1449, Dec. 1987.
[34] A. Seznec, "A case for two-way skewed-associative caches," *ACM SIGARCH Comput. Archit. News*, vol. 21, pp. 169–178, 1993.
[35] G. S. Sohi, "Logical data skewing schemes for interleaved memories in vector processors," Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep. #753, 1988.
[36] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *Proc. 11th Int. Conf. Supercomputing*, 1997, pp. 76–83.
[37] N. Topham, A. González, and J. González, "The design and performance of a conflict-avoiding cache," in *Proc. 30th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 1997, pp. 71–80.
[38] Mathworld. [Online]. Available: mathworld.wolfram.com/IrreduciblePolynomial.html
[39] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way cache: Demand-based associativity via global replacement," in *Proc. 32nd Annu. Int. Symp. Comput. Archit.*, 2005, pp. 544–555.
[40] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive GPU cache bypassing," in *Proc. 8th Workshop General Purpose Process. Using GPUs*, 2015, pp. 25–35.
[41] M. Mantor, "AMD Graphic Core Next Architecture," *AMD Fusion Developer Summit*, 2011.
[42] 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 *Proc. 18th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2013, pp. 395–406.
[43] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing," in *Proc. 29th ACM Int. Conf. Supercomputing*, 2015, pp. 67–77.

Mahmoud Khairy received the BSc and MSc degrees in computer engineering from Cairo University, Egypt. His research interests include GPGPU architecture, heterogeneous architecture, and emerging memory technologies.

Mohamed Zahran received the PhD degree in electrical and computer engineering from the University of Maryland, College Park. He is currently a faculty member with the Computer Science Department, New York University. His research interests include architecture of heterogeneous systems, hardware/software interaction, and biologically-inspired architectures. He is a senior member of the IEEE.

Amr Wassal received the PhD degree in electrical and computer engineering from the University of Waterloo, Ontario, Canada, in 2000. He has held several senior technical positions in the industry with Si-Ware Systems, PMC-Sierra, and IBM Technology Group. He is currently an associate professor in the Computer Engineering Department, Cairo University. He has a number of conference and journal papers and patent applications in the areas of multi-core architectures and their applications in DSP and sensor fusion. He is a member of the IEEE.