Access Pattern-Aware Cache Management for Improving Data Utilization in GPU

Gunjae Koo* Yunho Oh† Won Woo Ro† Murali Annavaram*
*University of Southern California
{gunjae.koo,annavara}@usc.edu
†Yonsei University
{yunho.oh,wro}@yonsei.ac.kr

ABSTRACT

Long latency of memory operation is a prominent performance bottleneck in graphics processing units (GPUs). The small data cache that must be shared across dozens of warps (a collection of threads) creates significant cache contention and premature data eviction. Prior works have recognized this problem and proposed warp throttling which reduces the number of active warps contending for cache space. In this paper we discover that individual load instructions in a warp exhibit four different types of data locality behavior: (1) data brought by a warp load instruction is used only once, which is classified as streaming data (2) data brought by a warp load is reused multiple times within the same warp, called intra-warp locality (3) data brought by a warp is reused multiple times but across different warps, called inter-warp locality (4) and some data exhibit both a mix of intra- and inter-warp locality. Furthermore, each load instruction exhibits consistently the same locality type across all warps within a GPU kernel. Based on this discovery we argue that cache management must be done using per-load locality type information, rather than applying warp-wide cache management policies. We propose *Access Pattern-aware Cache Management* (APCM), which dynamically detects the locality type of each load instruction by monitoring the accesses from one exemplary warp. APCM then uses the detected locality type to selectively apply cache bypassing and cache pinning of data based on load locality characterization. Using an extensive set of simulations we show that APCM improves performance of GPUs by 34% for cache sensitive applications while saving 27% of energy consumption over baseline GPU.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; **Single instruction, multiple data**;

KEYWORDS

GPGPU; cache management, memory access patterns

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080239

ACM Reference format:
Gunjae Koo* Yunho Oh† Won Woo Ro† Murali Annavaram*. 2017. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080239

1 INTRODUCTION

Graphics processing units (GPUs) exploit thread level parallelism (TLP) to maximize throughput with thousands of compute units. But GPU's memory hierarchy is designed for regular address accesses generated from vector-based operations such as those seen in multimedia applications. Hence, even though a warp or a wavefront (a collection of 32 threads) may generate many memory requests in practice it is expected that most of the individual thread requests can be coalesced into a single wide memory access. As such, GPUs are provisioned with 128 byte wide cache lines to enable a warp to fetch data from a single cache line. Traditionally GPUs have been provisioned with a small L1 cache which can accommodate only a few wide cache lines in each cache. For instance NVIDIA's Kepler architecture has 16KB of local data cache (128 cache lines), but allows up to 64 concurrent warps per core [30]. Theoretically each warp has just two lines worth of data that can be held in the local cache.

To reduce cache contention from multiple warps, warp throttling schemes have been proposed [9, 19, 26, 33]. These schemes essentially reduce the number of active warps contending for cache. However, in this work we show that each load instruction within a warp exhibits a specific type of data locality and would benefit from having a cache management that is tuned for that specific load instruction. As such warp-level cache management schemes, such as bypassing the cache for the entire warp, are generally too coarse since they cannot take advantage of per-load locality information.

To demonstrate this observation, Figure 1 shows the change in L1 data cache miss rate for the top four (based on dynamic access count) global load instructions in cache sensitive applications configured with the maximum TLP (48 active concurrent warps allowed per streaming multiprocessor) and the best case static warp throttling that limits the number of active warps that maximizes performance (CCWS-SWL-best) [33]. As can be seen in the figure, throttling warps has uneven impact on the cache miss rate reduction across the global loads. For the BFS benchmark, the miss rate of $Ld2$ is dramatically reduced with warp throttling, but the other three loads ($Ld1$, $Ld3$ and $Ld4$) see marginal reduction in miss rate. The reason for improved miss rate for $Ld2$ is that the accesses from that load

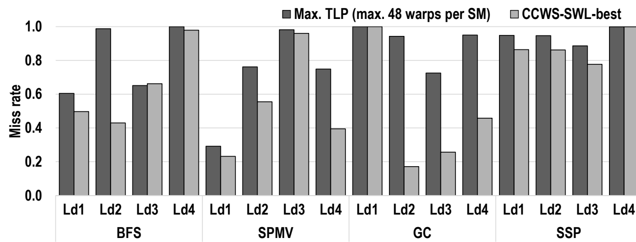


Figure 1: Miss rate change per load by warp throttling

exhibit strong temporal locality. But when dozens of active warps are accessing the cache this temporal locality cannot be exploited because requests from other warps evict the cache lines before temporal reuse. But with warp throttling the cache lines that were demand fetched by *Ld*2 stay in the cache longer, thereby improving the possibility for temporal reuse. On the other hand, the three other loads do not exhibit much temporal locality. In fact *Ld*4 exhibits streaming behavior. However, streaming cannot benefit from warp throttling, rather what is needed for *Ld*4 is to bypass the cache entirely which will eliminate unnecessary cache occupancy thereby allowing *Ld*2 to exploit temporal locality even further.

The above analysis shows that warp-based cache management does not exploit the per-load cache preferences. In other words, with per-warp cache management schemes cache space is still wasted by streaming data in *active* warps. On the same note, a warp-level cache bypassing scheme is also not appropriate since any load instruction that exhibits strong temporal locality is also forced to bypass the cache if the loads are issued from *bypassing* warps [26]. These observations motivate the need for a cache management scheme that uses the per-load locality behavior. Another interesting observation in GPU applications is that each global load instruction has a fairly stable behavior during the entire application execution time. Namely, whether a load instruction benefits from warp throttling or benefits from cache bypassing is independent of the warp ID or when that load is executed in the code. This property is based on the GPU's unique software execution model where all warps originate from the same kernel code. Hence, cache-preference properties of a load, such as data locality types or cache sensitivity of a certain load instruction detected in one warp can be widely applied to the same load execution in all other warps.

Based on these observations, we propose *Access Pattern-aware Cache Management* (APCM) for GPUs. APCM improves the data utilization in L1 data cache by devising per-load based cache management schemes. In particular, the following are the contributions of this work:

- We analyze the data access patterns in general purpose GPU applications and discover that individual load instructions in a warp exhibit four different types of data locality behavior: (1) data brought by a warp load instruction is used only once, which is classified as streaming data (2) data brought by a warp load is reused multiple times within the same warp, called intra-warp locality (3) data brought by a warp is reused multiple times but across different warps, called inter-warp locality (4) and some data exhibit both a mix of intra- and inter-warp locality.

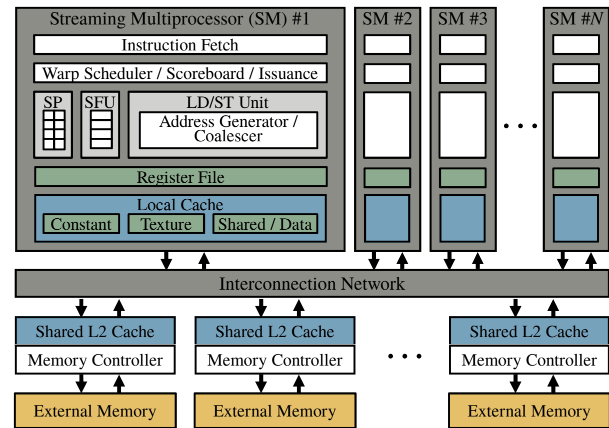


Figure 2: Baseline GPU architecture

- APCM applies locality-specific cache management scheme based on the detected locality type of a load instruction. APCM monitors cache access patterns from a single monitored warp, similar to the notion of Pilot Warp [3], and dynamically detects the load locality type. Based on the locality type APCM either protects a cache line fetched by a load, or bypasses the cache entirely for a load instruction. We design the microarchitecture structures and logic necessary for characterizing the locality type of each load and the associated runtime mechanisms for activating load-specific cache management policies.

The remainder of this paper is organized as follows. GPU memory hierarchy is discussed in Section 2. GPU cache access patterns are studied in Section 3. APCM algorithm and hardware are described in Sections 4 and 5. Evaluation details are in Section 6. Related work is discussed in Section 7. We conclude in Section 8.

2 BACKGROUND

2.1 Baseline GPU Architecture

Figure 2 shows the baseline GPU hardware architecture, which is based on the NVIDIA GPU design disclosed to the public [2, 29, 30]. While the NVIDIA's terminology is used here to describe GPU hardware the proposed design is broadly applicable to other GPU architectures, such as the AMD GPU architecture [4]. A single GPU is composed of multiple streaming processors (SMs) and shared L2 cache partitions connected via interconnection network. Each SM is equipped with streaming processors (SPs, also known as CUDA compute cores) that execute integer and floating-point instructions, special function units (SFU) for complex arithmetic instructions and load store (LDST) units. A group of 32 threads, called a warp, share the same program counter (PC). Warp is the basic execution unit issued to compute cores. Consequently, the number of the compute cores decides the maximum number of warps executed concurrently. For instance, Kepler has 192 compute cores per SM and thus can issue 6 warps simultaneously [30]. Issuance of warps is managed by warp schedulers which select a warp to be issued from a pool of ready warps, whose readiness is monitored by a scoreboard. As long as ready warps exist and execution units are available, an SM is able to execute instructions without stall.

Memory related instructions are managed in LDST units. An SM embeds different types of L1 caches (constant, texture and data) to deal with specific data spaces that are supported by the CUDA programming model [28]. Contrary to arithmetic instructions, which may exploit massive parallelism with more compute cores, only a limited number of memory instructions may access the cache at the same time. This limitation is due to the complexity of supporting multiple read/write ports on the memory structures. Thus memory requests generated from the 32 threads in a warp are coalesced to one or two requests if these requests all access contiguous data space of 64 or 128 bytes. Such coalescing hardware is common in GPUs. For instance, AMD GPUs have coalescing unit for vector load operation [4].

GPU caches are designed to have wide cache lines to maximize throughput of memory operations by concurrent threads, particularly in the presence of coalesced memory operations [13, 32]. However, in the presence of diverged memory operations a single warp may generate multiple narrow memory requests thereby causing tremendous cache pressure. It has been shown that L1 data cache lines are frequently evicted before they are reused due to the small number of cache sets per warp [26, 32]. The inefficient use of L1 data cache is discussed further in Section 3. If requested data is not found in local caches, fetch from the external DRAMs takes hundreds or even thousands cycles depending on data traffic in the interconnection network and memory channels [21, 42].

2.2 GPU Software Execution Model

GPU applications are composed of multiple kernels, which are massively parallelized tasks executable on GPU hardware. Each kernel is composed of multiple groups of threads called cooperative thread arrays (CTA) or thread blocks. The dimension of a CTA can be configured by programmers and is also limited by the GPU hardware. CTAs are allocated to SMs in a round-robin fashion until hardware resources such as a register file (RF) or shared memory in an SM is exhausted, whichever limit is reached first. The number of concurrent CTAs is also constrained by GPU specification (Fermi: 8 CTAs, Kepler: 16 CTAs) [29, 30]. Threads in a CTA are split into warps; warps originating from the same kernel share the same code and have similar characteristics [8].

3 CACHE ACCESS CHARACTERISTICS

In this section we study the cache access characteristics of several benchmarks selected from various GPU benchmark suites. We classify the selected benchmarks to three types - cache sensitive (CS), cache moderate (CM) and cache insensitive (CI). We use the following criteria for classification; the IPC of CS benchmarks is improved over 1.5× when 4× the baseline cache size of 32KB is used. On the other hand, the performance change of CI benchmarks is less than 10% when using the 128KB cache compared to the 32BK cache baseline. The performance of CM benchmarks falls in between the two extremes. The benchmarks studied in this paper are listed in Table 1.

Abbr.	Description
Cache Sensitive (CS)	
BFS	Breadth-First Search [7]
KMN	K-means [7]
IIX	Inverted Index [15]
WC	Word Count [15]
GC	Graph Coloring [41]
SSP	Single-Source Shortest Path [6]
Cache Moderate (CM)	
SPMV	Sparse-Matrix Dense-Vector Multiplication [36]
MM	Matrix Multiplication [15]
SS	Similarity Score [15]
CCL	Connected Component Labeling [6]
Cache Insensitive (CI)	
GE	Gaussian Elimination [7]
SRD	Speckle Reducing Anisotropic Diffusion [7]
MRI	Magnetic Resonance Imaging - Gridding [36]
SGM	Register-Titled Matrix-Matrix Multiplication [36]
STN	Stencil 2D [11]
APS	All Pairs Shortest Path [41]

Table 1: Benchmarks

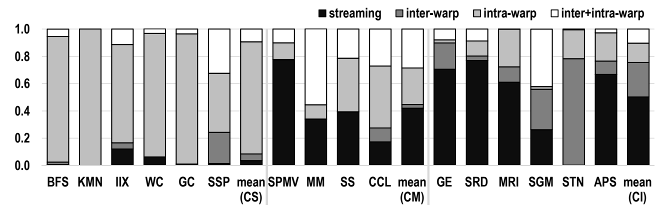


Figure 3: Ratio of data regions by data locality types

3.1 Data Locality Type

The locality exhibited by data fetched from each warp load instruction can be broadly classified into four types: streaming, inter-warp, intra-warp and inter+intra-warp locality. Streaming data is brought into the data cache on a demand fetch but is never reused. Hence, it has zero temporal locality. If the data fetched by a load instruction from one warp is also accessed by the same load PC across multiple warps, it is defined as inter-warp locality. If the data fetched by a load instruction from one warp is exclusively used within the same warp that data is supposed to exhibit intra-warp locality [33, 34]. The last category is when data is brought into cache by one warp and then repeatedly re-referenced by other warps as well as the original warp. Figure 3 shows the breakdown of all data accesses into the four categories. To generate the data in Figure 3, we simulated an infinite sized L1 data cache so as to get a fundamental understanding of how data is reused within and across warps, without worrying about cache replacement interference. The ratio of each type in the figure is computed as a number of cache lines (128 byte size) having a specific locality type divided by all allocated cache lines in the infinite cache.

Comparing the data presented in this figure with the benchmark categorization, it is clear that CI applications have a large fraction of streaming and some inter-warp locality type data. As such providing larger cache to these applications will not improve miss rate. Note that STN is categorized as CI in Figure 3, although it has some inter-warp locality. Further analysis revealed that most of this locality exists within a short time interval. Once a cache line is brought into the cache it is consumed in quick succession by multiple warps. As such there is no need to preserve cache lines for long time period

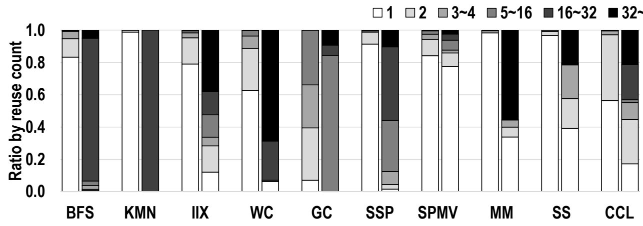


Figure 4: Ratio of number of cache lines by access count

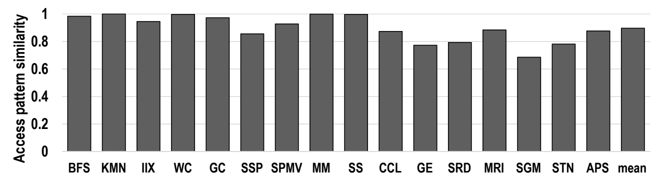


Figure 5: APS for infinite sized cache

and hence STN is categorized as CI. CS applications exhibit strong intra-warp locality, thus these applications have potential to reuse data in L1 cache if the data is not evicted before re-reference. CM applications have a mix of streaming and intra-warp locality type data, hence it is likely that the cache utilization suffers from the interplay of streaming and non-streaming data, where the streaming data may evict non-streaming data that may have temporal re-use.

3.2 Loss of Locality in Cache

The locality properties seen clearly with the infinite sized cache are completely obliterated when using a finite sized cache, when tens of concurrent warps share a small L1 cache. Figure 4 shows two stacked bars for each benchmark. Each category in the stacked bar shows the access count to a given cache line before it is evicted. The left stacked bar presents the data for a 16KB L1 data cache and the right bar is for an infinite sized cache. Looking at the 16KB data the results reveal that most of the cache lines are accessed only few times (in fact just once or twice). This data may give the impression that the data has limited reuse although our earlier results showed that there is plenty of reuse in the cache sensitive applications. The main reason for the dramatically different view seen on the 16KB cache is that the data locality that exists for a given load in one warp is severely perturbed by tens of other warps running concurrently. Thus frequent cache line eviction is the biggest culprit that hides the fundamental data sharing behavior that is prevalent in many of benchmarks. This fact is evident when looking at the right stacked bars which simulates an infinite data cache. When the cache size is infinite there is no eviction and hence much of the intra-warp locality that was seen in Figure 3 is clearly manifested in the results. This data implies that identifying the locality types correctly is not feasible with the small sized local cache since sharing behavior is frequently lost due to severe thrashing.

3.3 Access Pattern Similarity

As threads originating from one kernel share the same program code, it is intuitive to expect that the same load instruction executed in different warps in an SM exhibit similar data access behavior. To investigate this intuition we use the notion of access pattern similarity (APS) exhibited by a load instruction across multiple warps in a given kernel. APS is quantified using Equation 1.

$$APS = \frac{\Sigma_i max(N_i^t \text{ of } Ld_i)}{\Sigma_i (N_i \text{ of } Ld_i)} \quad (1)$$

N_i is the number of unique cache lines requested by a load Ld_i. Thus the denominator is simply the sum of the number of unique cache lines accessed by all loads in a given kernel. Then for each load i that brings a given line into cache we categorize that cache line into one of the four warp-based locality types discussed in Section 3.1. N_i^t of Ld_i means the number of cache lines whose access patterns fall into access locality type t. We then compute the maximum across all categories t to find $max(N_i^t)$ for each Ld_i. Thus the numerator shows how dominant is a particular access type for each load. APS values closer to one indicate each load instruction has a dominant locality type across various warps executed in the kernel. Consequently, APS quantifies consistency of access patterns of a load instruction in a kernel.

Figure 5 shows the APS of the benchmarks run with an infinite L1 data cache configuration. Overall, most applications, especially cache sensitive applications, have an APS value close to one. The average APS for all tested applications is 0.90. Thus we conclude that each load in a GPU application has a consistent cache access pattern across all warps.

4 ACCESS PATTERN-AWARE CACHE MANAGEMENT

In the previous section we showed that data fetched exhibits one of four dominant locality types. Furthermore, the load instruction that fetches a particular locality type data tend to fetch the same locality type of data for the entire kernel execution. Based on these observations we propose *Access Pattern-aware Cache Management* (APCM). APCM exploits the observation that each load exhibits a persistent data locality type to improve utilization of GPU L1 data cache. APCM first detects a data locality type of each load instruction by monitoring cache access patterns of one exemplary warp. Since data locality type cannot be inferred from a regular cache due to severe interference in the cache from multiple warps, APCM uses a dedicated cache tag array to track data sharing behavior from only one warp. The locality type inferred for each load in the monitored warp is then applied for the same load across all warps within a kernel with confidence since the cache access patterns exhibit strong consistency among warps as shown in Section 3.3. APCM then applies load-specific cache management scheme for each data locality type as described below.

4.1 Locality-Specific Cache Management Strategies

As stated earlier, data requested by global loads exhibits load-specific locality patterns, and these loads have strong access pattern similarity where the load exhibits a given locality type across all warps and for the entire kernel execution time. Since effective lifetime of cache lines is characterized by data locality we argue that locality-specific cache management strategies are desired for each load instruction.

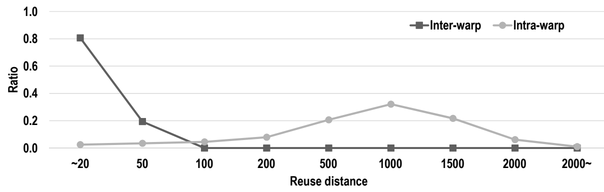


Figure 6: Reuse distance for different locality type data

Streaming data: As shown in Figure 3, streaming data occupies significant fraction of demanded data in CM and CI applications. That means resources like cache lines and MSHR entries are wasted when streaming data is fetched into the cache. To make matters worse even if there are a few cache lines with strong locality in these applications they may be evicted by the streaming data. Thus the best way to treat streaming data is to bypass the L1 cache entirely and provide data to compute cores directly from the L2 cache and its associated interconnection network.

Inter-warp locality: Using a combination of address traces and source code analysis we discern that the primary reason for inter-warp locality is stride accesses to large data arrays across different warps. The index of the data array accessed by each thread is typically computed using a linear function of thread IDs and/or CTA IDs [22, 35]. Threads' data addresses are merged into one cache line space, however coalesced requests straddle two cache line regions if addresses are misaligned. In that case a single warp access may bring in two cache lines, but only a part of second cache line is accessed by the current warp. The neighboring warp will access the leftover data in the second cache line and then fetch a new cache line which is again partially used. Such misaligned data accesses cause inter-warp sharing. The other reason of inter-warp locality is small data request size. Even if threads' request addresses are merged, small data size, for example 1 byte per thread, cannot occupy an entire cache line. Then the neighbor warps consume the remaining part of the cache line space. Such inter-warp locality among neighbor warps or thread blocks has also been observed before [21, 23].

Figure 6 shows distributions of the reuse distance, which is defined as the number of cache accesses until a cache line is reaccessed, for inter-warp and intra-warp locality type data found in SGM and BFS, respectively, with infinite sized cache. As GPUs interleave warps in quick succession data that is accessed by neighboring warps is in fact accessed in a short time interval, and hence inter-warp locality loads have a short reuse distance. Using a simple LRU policy keeps most of the inter-warp locality data in cache and hence, no specific cache management approach is necessary for inter-warp locality type as long as neighboring warps are scheduled in quick succession, which is generally the case with round-robin scheduling policies.

Intra-warp locality: Intra-warp locality is the dominant type for CS applications. Cache lines allocated by loads of the intra-warp locality type are not efficiently reused, even though they are referenced multiple times. Figure 6 shows that intra-warp locality type has long reuse distance, which is a result of GPU warp schedulers that interleave warps; even instructions that are close-by in a warp are effectively separated by a large time interval. As such intra-warp locality type data suffers frequent interference by many accesses from other warps leading to premature eviction of cache lines. We propose to exploit this data locality by protecting the cache lines allocated by loads of intra-warp locality type until they are mostly done with their reuse. Details of the process will be explained shortly.

4.2 Detection of Locality Types

APCM first detects a locality type per load before applying the specific cache management policy. The locality types of cache line data can be detected based on access counts by the same or different warps. For instance, if a cache line is first allocated by warp *A* and then requested by warp *B*, then the total access count for the cache line becomes two and the access count from the warp that initially allocated the cache line (warp *A*) is one. This cache line can then be inferred to exhibit inter-warp locality. Locality detection criteria by APCM is summarized in Table 2.

Locality type	Total access count	Access count by allocating warp
Streaming	1	1
Inter-warp	N (> 1)	1
Intra-warp	N (> 1)	N
Inter+intra-warp	N (> 1)	M (< N)

Table 2: Criteria of locality type decision

However, detecting locality types by keeping track of the access counts to a cache line is difficult in GPUs. Access profiles stored in cache tags are frequently lost as cache lines are evicted. In order to solve this challenge APCM tracks access patterns of one warp, called a monitored warp, with a small tag array structure, called monitor tag array (MTA). MTA works like a private cache tag array that tracks the accesses from a monitored warp. Each tag in the MTA is augmented to collect total access counts and access counts by the monitored warp. The detected locality types by the monitored warp can be applied to other warps with high accuracy since locality types are consistent among warps from the same kernel as mentioned in Section 3.3. The microarchitecture details of MTA are described shortly.

4.3 Protection Algorithm

As explained in Section 4.1 APCM applies cache bypassing and cache line protection for data allocated by load instructions detected as streaming and intra-warp locality types respectively. Cache bypassing can be simply implemented by allocating demand requests to the injection queue of the interconnection network without perturbing the L1 data cache. On the other hand, cache line protection requires pinning a cache line for the lifespan of data (from the first allocation to the last access) for optimal cache resource utilization. Counter-based or reuse distance-based cache line protection algorithms have been explored in CPU domains, however those methods are ineffective for GPU's due to concurrent and interfering accesses from dozens of warps. As shown in Figure 6 reuse distance for intra-warp locality type data is widely distributed, thus it is difficult to estimate effective protection distance.

In order to estimate accurate lifespan of cached data, APCM tracks data access dependency between load instructions. If a certain cache line is first allocated by load *A* and then re-referenced lastly by

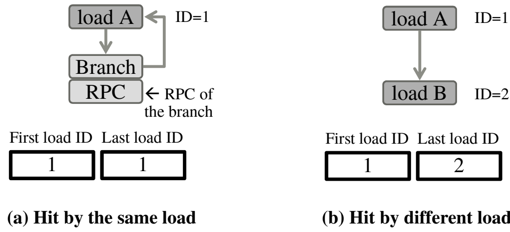


Figure 7: Load dependency and consumer load ID

load *B*, the lifetime of the corresponding cache data can be estimated between execution of load *A* and *B*. Therefore, APCM also tracks the IDs (hashed PC) of the first allocating load and the last accessing load for the monitored warp in MTA tags, and then exploits this dependency information to estimate the lifetime of the protected lines by all other warps (more details in the next section).

Figure 7 shows two examples of load access dependency scenarios. The left-hand case represents the allocated cache line is reaccessed by the same load executed in a loop. For this case the hashed PC of load *A* is logged in *First load ID* field in an MTA tag and then the same hashed PC is stored in *Last load ID* field. Consequently, APCM estimates the valid life of the cache lines allocated by load *A* ends when the loop is escaped. The right-hand case shows the cache line is re-referenced by the different load instruction. The logged IDs in the first (load *A*) and the last (load *B*) load ID fields are different in this case, then APCM disables protection of the cache line after executing load *B*.

5 HARDWARE ARCHITECTURE

The hardware architecture of APCM is described in Figure 8. The modifications are primarily made to the L1 data cache access pipelines in LDST units. The additional structures added in the figure are: monitor tag array (MTA), cache access information table (CAIT), load alias table (LAT) and protection status board (PSB). As a brief

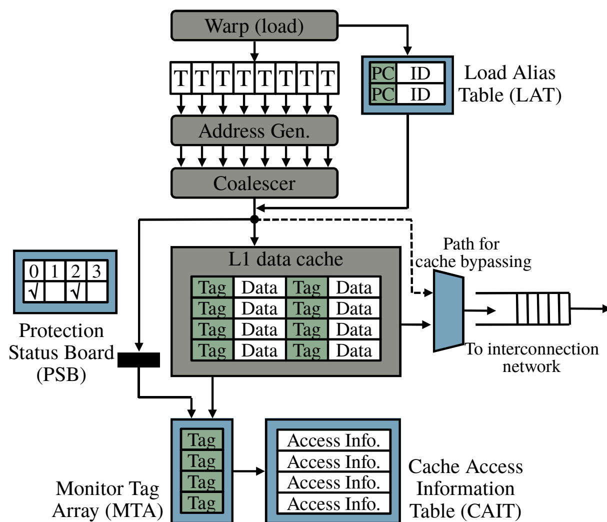


Figure 8: Hardware architecture of APCM

overview of the proposed hardware, MTA is a tag array structure to track access count and data access dependency for the monitored warp. CAIT manages the detected locality types per each load instruction. LAT converts 32-bit PC address of load instructions to a shorter hashed ID, primarily to save storage space in other structures tracking the per load information. PSB maintains information on which warps are currently protecting cache lines.

5.1 Tracking Access Patterns

APCM tracks the access history of just one monitored warp in MTA. Figure 9b shows the structure of one entry in MTA. There are four additional fields in each tag entry alongside the usual tag information. First and last load ID fields store which load instruction first allocated a given cache line and the last load instruction that accessed that cache line. The access count field stores the total number of time a given cache line is accessed by any warp (including monitored warp), and intra-warp access count tracks how many times the monitored warp alone accessed that cache line.

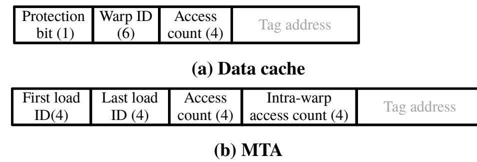


Figure 9: Additional fields in tags

The process for accessing MTA is as follows. When a global load instruction is first executed, the load PC is hashed to create a shorter load ID. The load PC and the load ID are then stored in a 16-entry content-addressable memory (CAM) of load alias table (LAT). If the load has already been executed at least once before, the LAT will have an entry for that load PC and that load ID is retrieved. Normally a GPU kernel contains small number of global loads, thus tracking the first 16 global loads is enough to capture nearly all the global memory accesses in a kernel. Results from the LAT size sensitivity study in the next section verifies this claim. For simplicity, if the LAT is full then only the first 16 loads are tracked and the remaining loads are treated as normal loads. If the load instruction originates from a monitored warp then the load address is used to generate an index into MTA. MTA works like the cache tag array. When a memory request from the monitored warp misses in MTA, a tag address of this request is allocated in MTA. Also the load ID (hashed PC of the load instruction) is logged in the *First load ID* field, and the *Access count* field and *Intra-warp access count* field are set as one respectively. If this tag is hit by other requests after allocation then the load ID of the requests is stored in the *Last load ID* field. Future accesses to this cache line from the monitored warp increment both the *Access count* and *Intra-warp access count* fields. Only the *Access count* field is incremented if load instructions from all other warps (other than the monitored warp) hit the MTA entry. If a single warp load generates more than two memory addresses (uncoalesced loads) then only the first two requests can allocate the MTA tag while the other accesses from the same load are simply dropped.

It is also possible that a request from the monitored warp hits in L1 cache and misses in MTA. That situation occurs when the cache

line is first allocated by any warp other than the monitored warp, and then the request from the monitored warp hits in the cache. In that case the access count from the L1 cache tag is used to initialize the MTA tag. To support this case, L1 cache tag is augmented to track access count (*Access count* field) as shown in Figure 9a, which simply tracks the number of times that cache line is accessed by any warp. Other two fields (*Protection* and *Warp ID*) are used for cache line protection, which will be explained later.

Valid bit (1)	Management method (2)	Last load ID (4)	Access count (4)

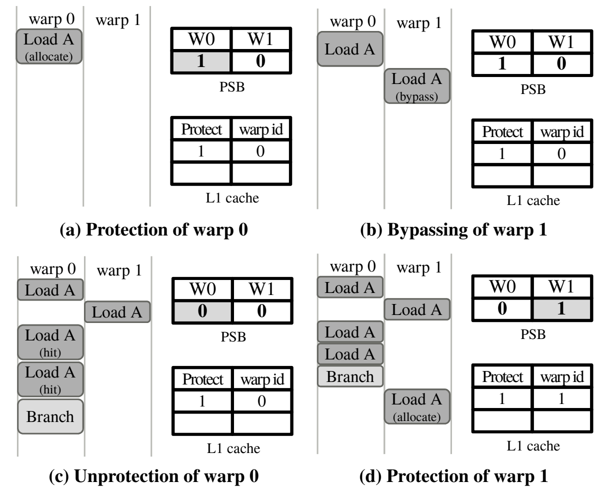
Figure 10: Fields in an entry of CAIT

CAIT manages the tracked data locality type and data access dependency information per load. Basically an entry of CAIT is updated when an MTA tag is evicted due to address conflicts, or the access count of an MTA entry exceeds the predefined threshold value. Figure 10 shows the content of a CAIT entry. Each entry of CAIT is indexed by the load ID stored in the *First load ID* field from the MTA entry. The *Management method* field stores the cache management scheme that is selected for the corresponding load instruction. The management field uses access count and intra-warp access count fields from the MTA entry to determine the load locality type using the criteria defined in Table 2. For streaming data the management method is set to *bypassing*, and if the load exhibits intra-warp locality the management method is set to *protection*. Otherwise the *normal* cache management scheme is set. The *Access count* and *Last load ID* fields are just copied from the MTA tag.

It is possible that after an MTA tag associated with a load ID is evicted the same load ID may execute again from the monitored warp and reallocate a new MTA entry. Thus before the monitored warp completes execution the MTA entry may potentially be evicted and allocated multiple times by the same load. Note that this is not a common scenario, but it is just a corner case scenario that must be handled. Since the CAIT entry is indexed using only the load ID on each MTA entry eviction the CAIT entry must also be updated. We use the following simple policy. An entry of CAIT is overwritten by the new information if the *Access count* value in an MTA entry is larger than the current *Access count* stored in the CAIT entry. Finally, after a monitored warp finishes execution all the MTA entries are scanned and each MTA entry updates the CAIT as described above. Then all the MTA entries are invalidated and may be used again for monitoring a different kernel execution later.

5.2 Cache Management

The overall cache management works as follows. A load instruction uses the load ID to index into a direct mapped CAIT. If the CAIT entry indexed by the load ID is valid, the cache management scheme as specified in the *Management method* field is applied for that load. Possible cache management methods are *normal*, *bypassing* and *protection*. If the management method specifies *normal*, the load goes through the normal GPU cache access process. If a load is categorized as *bypassing* type, requests from the load are directly assigned to the interconnection without accessing L1 cache. If the load access requires protection, the allocated cache line is pinned as a protected line by setting the *protection* bit of the cache tag (presented in Figure 9a) in L1 cache. In addition, the *Warp ID* field



(a) Protection of warp 0
(b) Bypassing of warp 1
(c) Unprotection of warp 0
(d) Protection of warp 1

Figure 11: Cache data protection control

of the corresponding cache line is also set to the current warp ID of the load instruction. Finally, we need a mechanism to determine when to unpin the protected cache lines.

The validity of the protected lines is controlled by protection status board (PSB). PSB has one entry per warp. When a load instruction in a warp allocates a protected cache line then the PSB bit for that warp is set to one. The *Last load ID* from CAIT is copied into the PSB entry for the warp. From then on PSB tracks if the load instruction mapped into the last load ID has completed execution at which time the PSB bit is reset to zero, which means the cache lines for that warp are no longer protected.

If the last load ID of a CAIT entry is identical to the load ID used in indexing that CAIT entry, the loop indication bit of the PSB entry is set in order to mark the load instruction as being part of an iterative loop. For this case the protected lines are pinned until the warp escapes the loop. We use the SIMT stack [14] to track when a load instruction exits the loop. In our implementation the PSB tracks only a single protected load instruction per warp. Our empirical analysis showed that in any given warp only a single load instruction needs to be protected in practice. Hence, if the PSB entry was already set by a former load instruction in a certain warp, protection control for other load instructions are simply ignored in that warp until the PSB entry is released. In practice, this simple approach works well across all the applications studied.

5.3 An Illustrative Example

An example of cache line protection is depicted in Figure 11. This example assumes that a monitored warp has completed execution and data fetched by load instruction (Load A) has been determined to be accessed multiple times within a loop during the monitored warp execution. The MTA entry accessed by the address from Load A would have marked the *First load* and *Last load* fields to be the same load ID, and the *Access count* and $Intra-warp\ access\ count$ fields are also the same at the end of the monitored warp execution. Based on this MTA entry information the CAIT entry would be updated to indicate that the Load A is characterized as an intra-warp

locality load, based on the criteria set in Table 2. The management method for the CAIT entry indexed by the Load *A* would have been set to be protection based for intra-warp loads.

In this example there are two warps. First, warp 0 executes Load *A*. The CAIT entry indexed by that load ID is valid and its management method is set to protection. At this time the the PSB of warp 0 is set to 1, indicating the data fetched for Load *A* into the L1 cache must be protected. Once the data is fetched in L1 cache the *protection* bit in the cache line is set to one and the *warp id* field of the cache line is set to warp 0. From then on warp 0 will continue to protect the cache lines fetched by Load A until that load exits the loop.

Assume the L1 cache is direct-mapped and warp 1 then executes Load *A*. Since CAIT is indexed only by load id, and is warp independent, the CAIT entry for Load *A* from warp 1 would have also indicated that the load requires protected cache lines. The Load *A* from warp 1 would try to fetch data into the cache line but notices that the cache line's protection bit is already set to one. The warp id is set to warp 0 for that protected cache line, and the PSB entry for warp 0 is still set to one, namely warp 0 has not exited the loop. Rather than evict a protected cache line, the data fetched by Load *A* from warp 1 would then simply bypass the cache as shown in Figure 11b.

The bit for warp 1 in PSB is not set since the request from warp 1 was not pinned. In Figure 11c the repeated Load *A* from warp 0 hits the protected cache line multiple times and eventually the loop terminates. Then the bit for warp 0 in PSB is reset since the lifetime for the protected cache line ends.

At this time if warp 1 is still executing Load *A* in loop it will continue to attempt to allocate a cache line and protect it on each execution of the load. After warp 0 exits the loop warp 1 may see the cache line to be protected. But warp 0 stored in the cache line is used to access PSB and eventually PSB indicates that warp 0 is no longer protected. At this time the cache line protection bit is reset and the warp 1 allocates that cache line and then sets its *protection* bit and sets the warp id to warp 1. The PSB for warp 1 is now set to one as shown in Figure 11d.

5.4 Hardware Cost

Extended in L1D	11 bits per tag, 128 lines
MTA	36 bits per tag, 32 lines
CAIT	11 bits per entry, 16 entries SRAM
LAT	32 bits per entry, 16 entries CAM
PSB	6 bits per warp

Table 3: Hardware overhead by APCM

Table 3 summarizes the required hardware resources for implementation of APCM. We use *CACTI* 6.5 to estimate area overhead and power consumption of the memory components based on 45 nm technology node parameters [27, 38]. The hardware cost of CAM in LAT is estimated based on the results in the published work [43]. Other parts are modeled in RTL level and synthesized with 45 nm FreePDK library to estimate area and power information [1]. The occupied area required by APCM is about 4700 μm^2, equivalent to 0.14 % of the L1 data cache area estimated by CACTI. It is also estimated that APCM increases 0.02 % of one SM area (22 mm^2), measured based on the die photo of GF100.

Parameter	Configuration 1	Configuration 2
SMs	15 SMs @ 1400MHz	16 SMs @ 876MHz
SIMT width	32	32
Warps / SM	48	64
CTAs / SM	8	16
Scheduler	LRR, 2 per SM	LRR, 4 per SM
CUDA cores	32 per SM	192 per SM
Register file	128KB	256KB
L1 data cache	16KB, 128B / line, 4-way, LRU, 64 MSHRs	
L2 cache	768KB, 8-way	1536KB, 16-way
DRAM	384b @ 924MHz	384b @ 1750MHz
GDDR5 Timing [16]	t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=5.5, t_{WR}=12, t_{CL}=12, t_{CL}=4	

Table 4: GPGPU-Sim baseline configurations

Parameter	Configuration
MTA	32 entries, direct-mapped
LAT & CAIT	16 entries
Cache Mgmt.	**b**ypass / **p**rotection

Table 5: Basic APCM configuration

6 EVALUATION

6.1 Methodology

We implemented APCM on *GPGPU-Sim* v3.2.3 [5]. The baseline configuration settings used for evaluation are listed in Table 4. Configuration 1 is similar to NVIDIA Fermi (GTX480) [29]. We use the first configuration as the baseline settings for the most part of evaluations. Additionally, we also use Configuration 2, which is similar to NVIDIA Kepler (GTX780) [30], to test the performance of APCM on the more recent architecture settings equipped with more compute cores, higher DRAM bandwidth and the same size of the L1 data cache. Although APCM is evaluated on GPGPU-Sim with NVIDIA GPU architecture, AMD GCN architecture also has similar data fetch process for the vectorized global loads as stated in Section 2, thus we believe our APCM approach is also applicable to other GPGPU architectures.

The basic configuration of APCM is shown in Table 5. MTA is a direct-mapped tag array with 32 entries. Depth of both LAT and CAIT is set as 16, therefore, data requested by 16 different global loads is tracked and managed. We studied the impact of using just cache bypassing, or just protection or a combination of both with APCM to isolate the performance impacts of each cache management scheme.

We used the benchmarks introduced in Section 3 to evaluate APCM. We simulated all applications until the end of their execution or when the executed instruction count reached one billion. One exception is for KMN, which was simulated until the completion of the first kernel execution because trailing kernels have only texture loads accessing the texture cache.

6.2 Performance

Figure 12 shows the performance of APCM normalized to the baseline Configuration 1. We tested the cache management methods by applying bypassing and cache line protection in isolation and then combined both. The protection based approach works well for the cache sensitive (CS) applications since intra-warp locality type data

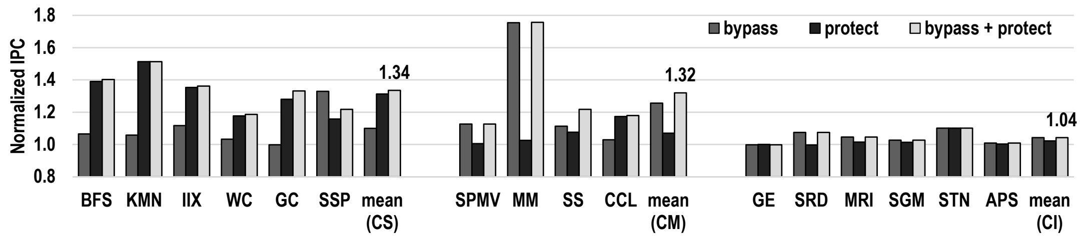


Figure 12: Performance of APCM normalized to the baseline Configuration 1

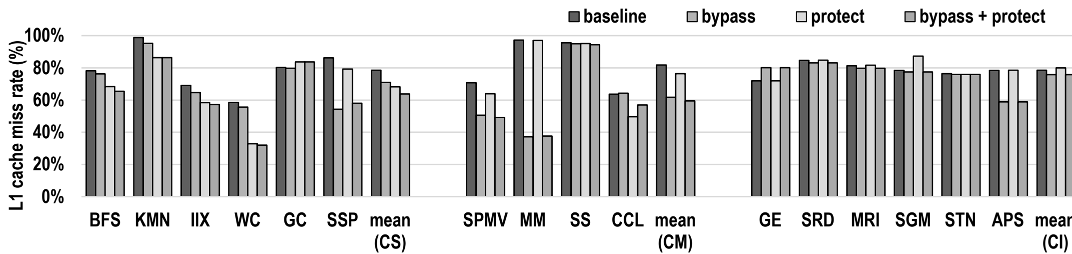


Figure 13: L1 cache miss rate

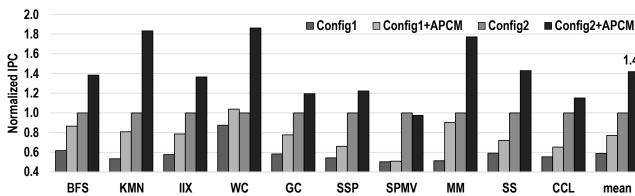


Figure 14: Performance on different GPU configurations

is dominant and effectively tracked by APCM for these applications. But for cache moderate (CM) applications the protection mechanism alone is not effective, however when combined with bypassing APCM works well. It is because the CM applications have mixed locality types of data, thus bypassing streaming data in L1 cache is helpful to keep other reused cache lines. Especially MM exhibits large ratio of intra-warp locality type data with the infinite cache model as shown in Figure 3, however the reuse distance of this intra-locality type data is too long to track in MTA. For this case it is detected as streaming data by APCM. Bypassing such type of data is better for performance since L1 cache cannot keep those cache lines even if only one warp is allowed to access the cache. For cache insensitive (CI) applications the performance is not degraded with APCM. Overall, when both bypass and protection are applied the performance of CS applications increases by 34% on average. The performance of CM applications is improved up to 76% for MM and the average performance is improved by 32%. IPC of CI applications is slightly increased (4%) by APCM since some data exhibits streaming patterns which were bypassed, and as a result some shared cache lines were kept longer before eviction. Overall, the average performance improvement achieved by APCM across all applications is 22%. Thus applications that are bottlenecked by cache see significant performance improvements, while applications that do not rely on cache do not suffer any performance degradation.

Performance of APCM was also evaluated using Configuration 2 which has more compute engines. Figure 14 shows the IPC of two base configurations and combinations of APCM normalized by the baseline Configuration 2. With the Configuration 2 settings enabling more massive TLP and higher DRAM bandwidth, the baseline performance is increased by 69% on average compared to the Configuration 1. Even with a much stronger baseline that can improve overall performance of applications, APCM can still enhance the performance with better utilization of data cache. APCM improves the performance of Configuration 2 by 42%, which is better than the enhancement for Configuration 1 (33% for CS+CM). Due to space constraints we did not show the CI benchmarks but they suffered no degradation in performance. This means APCM effectively resolves cache contention in more compute-intensive hardware like Kepler.

6.3 Cache Efficiency

Figure 13 shows the miss rate of the L1 data cache by APCM. The miss rate of CS applications is reduced by on average 15% compared to the baseline configuration 1, since cache lines allocated to intra-warp locality type data are reused more effectively. As the fraction of other locality type data is very low for CS applications, impact of cache bypassing is insignificant. On the other hand, cache bypassing for the streaming data is effective for CM applications since other shared cache lines can remain in the data cache without interference from the streaming data. For CM applications, the cache miss rate is reduced by 22% with cache bypassing of APCM compared to the baseline. Streaming data is dominant for CI applications and memory requests of this type bypasses the cache with the bypassing scheme of APCM. However, the access pattern similarity, shown in Figure 5, is not high for some CI applications. When access pattern similarity metric is low, as is the case for a few CI benchmarks, then using a single monitored warp to detect load access patterns is not accurate. As a result some loads may be misclassified in CI applications. Hence, even though the overall miss rate dropped some of the misclassification reduced the potential performance benefits for CI applications.

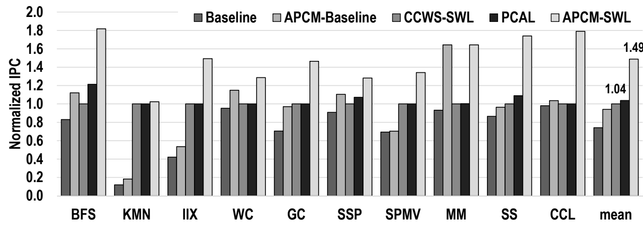


Figure 15: Performance with warp throttling methods

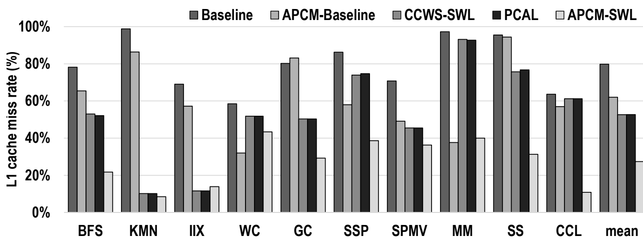


Figure 16: L1 cache miss rate with warp throttling methods

6.4 Performance with Warp Throttling

Limiting the number of active warps is one of the ways proposed in the literature to alleviate cache thrashing and congestion in memory systems by blocking issuance of memory requests from inactive warps. In addition the state-of-the-art warp control schemes exploit cache bypassing from several warps in order to make use of underutilized memory system resource when warp throttling is applied. APCM also can be applied on top warp throttling methods to maximize utilization of cache resource by applying fine-grained per-load cache resource control. In order to investigate the performance impact of the per-load cache resource control by APCM, we implemented APCM on top of warp throttling and compared its performance to the state-of-the-wart warp-level cache management method.

Figure 15 compares the performance of warp throttling approaches and APCM. IPC of each approach is normalized to the performance obtained by the best static warp throttling configuration (CCWS-SWL). The best static throttling scheme outperforms dynamic warp throttling methods [33]. PCAL is the priority-based warp-level cache allocation method presented in [26]. We compare the performance of the static PCAL since it shows better performance than other dynamic approaches. We implemented APCM on the baseline configuration (Configuration 1) allowing maximum active concurrent warps (APCM-Baseline) and then applied APCM on top of static warp limiting method (APCM-SWL).

APCM shows better performance improvement than prior warp throttling schemes and in fact APCM can be applied orthogonally on top these warp throttling techniques. On average APCM-SWL outperforms PCAL by 43% and CCWS-SWL by 49%. It is because the per-load cache management of APCM utilizes cache resource more efficiently by applying selective bypassing or pinning based on the detected locality types. Figure 16 compares L1 cache miss rate of each approach, and it reveals lower cache miss rates for most applications when using APCM. In addition APCM with the warp throttling can allow more TLP since fine-grained cache management by APCM alleviates cache contention resulting from more concurrent warps. Hence more active warps are allowed with APCM-SWL compared to CCWS-SWL.

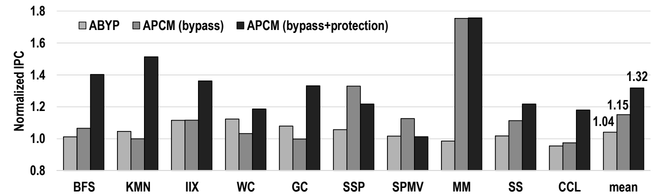


Figure 17: Performance by bypassing methods

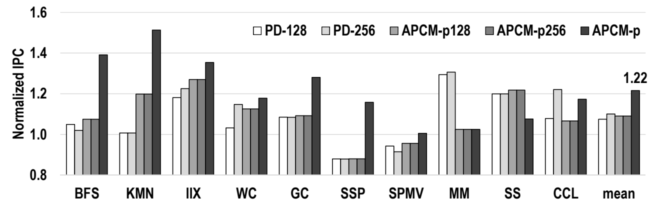


Figure 18: Performance by protection methods

6.5 Comparison with Other Schemes

Cache bypassing: We compared the performance of APCM with the adaptive GPU cache bypassing (ABYP) presented in [37]. ABYP monitors the evicted cache lines and collect this information in a per-load table. ABYP predicts the future requests from the load which will not be reused if cache lines allocated by a load are evicted multiple times without re-reference. Even though ABYP exploits L2 cache to avoid false decision of streaming data, it can still mislabel data with strong locality as streaming data due to limited L1 cache size. Hence, using a monitored warp to track this information in MTA is critical. Figure 17 shows the performance comparison results. We first compare ABYP with APCM-bypass only. The performance of ABYP is comparable to APCM-bypass only for CS applications. In fact for a few applications ABYP sees better performance because it may aggressively characterize some data as streaming which will be bypassed, thereby enabling some of the intra-warp data to stay in cache longer to improve hit rate. Overall, the cache bypassing scheme used in APCM improves performance by 15%, which is 11% better than ABYP. However, when the full APCM scheme is enabled APCM significantly outperforms ABYP by more than 28%.

Cache line protection: APCM estimates effective data lifetime in cache based on load data dependency. We compared the performance with other cache line protection schemes as shown in Figure 18. PD means the static reuse distance based cache line protection method [12]. The number following PD represents protection distance. With PD based cache protection, data in the cache line is kept until access count for the cache exceeds the defined reuse distance after the cache line is allocated. We also tested the static protection distance scheme for APCM, where only the cache lines allocated by the load instructions determined as *protection* bit in CAIT (APCM-pN, where N is a protection distance). APCM-p applies our protection scheme based on load data dependency. In the figure, all IPC values are normalized to the baseline Configuration 1. The average performance improvement by our scheme is 22%, which is

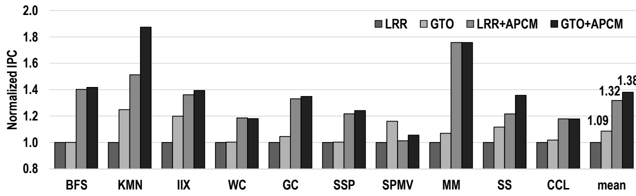


Figure 19: Performance by warp schedulers

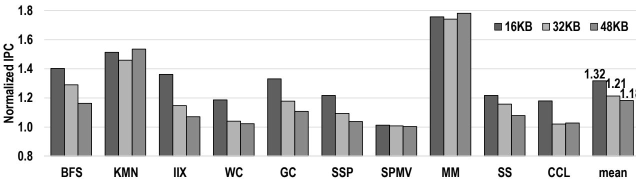


Figure 20: Performance by L1 cache size

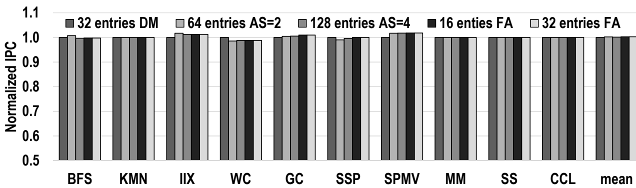


Figure 21: Performance by MTA configurations

better than static protection distance schemes. Since APCM protects cache data based on data dependency as described in the previous section, it can predict lifetime of cache lines more effectively.

6.6 Sensitivity Studies

Warp scheduler: Figure 19 shows the performance impact of using different base warp schedulers, loose round-robin (LRR) and greedy-and-oldest (GTO), with APCM. Normally it is known that GTO warp scheduler is more effective for memory-intensive applications since it allows a single warp to continue to execute independent instructions to hide memory latency. Figure 19 reveals that GTO improves performance by 9% for CS and CM applications. But APCM is effective even on top of GTO and improves performance by 27% over GTO scheduler baseline.

Cache size: Modern GPU architecture provides flexible options for programmers to configure the size of L1 data cache as 16KB, 32KB and 48KB [29, 30]. Figure 20 compares the performance improvement by APCM normalized to the different L1 cache size baselines (16KB, 32KB and 48KB per SM). For CS and CM applications APCM improves the performance by 21% and 18% on average with 32KB and 48KB L1 cache respectively. Note that the baseline performance is significantly enhanced with larger L1 cache configurations for the cache sensitive applications. This evaluation result reveals that APCM is effective even if larger L1 cache is applied.

MTA configuration: As MTA is exploited to track the cache accesses for the monitored warp, a large size of MTA may capture more access history. Figure 21 shows the performance change of APCM with various MTA configurations. The performance by all configurations is normalized to the performance of basic MTA configuration with 32 entries shown in Table 5. "DM" means direct-mapped tag array and "FA" means full-associated tag structure. "AS=*N*" represents N-way-associative tag structure. Across a range of MTA sizes with varying degrees of associativity the performance improvements are all within 5% of the baseline MTA configuration.

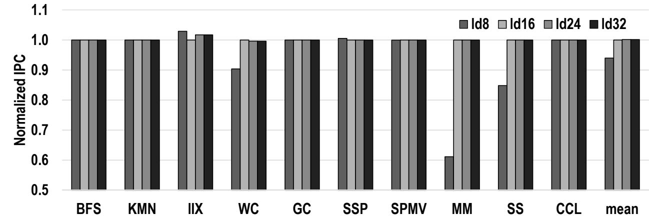


Figure 22: Performance by LAT and CAIT depth

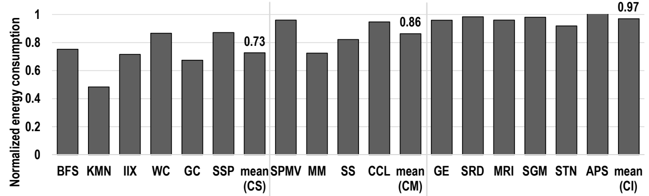


Figure 23: Normalized energy consumption

LAT and CAIT depth: The depth of LAT and CAIT decides the number of different global load instructions that can be managed by APCM. We varied LAT and CAIT sizes and the results are shown in Figure 22. The number of "ld" means the number of different loads managed by APCM. Performance metric (IPC) is normalized to the basic APCM configuration (ld16). The result reveals the performance of APCM saturates when the depth is set at 16. Normally a GPU kernel has a small number of global loads and hence tracking about 16 such loads seems to capture most of the benefits of APCM.

6.7 Energy Consumption

Figure 23 shows energy consumption reduction by APCM normalized to the baseline Configuration 1. As discussed in Section 5.4, power and energy consumption of APCM is estimated with CACTI and synthesized RTL models under the 45 *nm* technology node. Energy and power consumption of other components of GPU are estimated with *GPUWattch* [24]. For KMN, APCM consumes about 48% of energy compared to the baseline machine. Some energy saving are observed even for CI applications that only benefit from bypassing of streaming data, however, the impact of bypassing is not large because energy consumption of L1 data cache is a small portion of whole GPU energy consumption [24].

7 RELATED WORK

GPU Cache Management: Efficiency of memory hierarchy is one of critical performance factors especially for general purpose applications with irregular data access patterns. Several studies have tackled the efficiency issues of memory subsystem and cache management for GPUs. Some studies exploited software or compiler based methods for GPU cache management. Choi et al. [10] apply write buffering and read-bypassing for shared cache to increase utilization of shared data regions and prevent cache pollution from unnecessary requests. Jia et al. [18] presented the compile-time

technique to characterize access patterns and data locality of GPU applications and then configure cache usage of the applications. Xie et al. [39] presented the compiler-based method to partition global loads into cached or non-cached by analyzing data reuse and memory traffic. Xie et al [40] also represented the software and hardware coordinated cache bypassing method. They use the compile time analysis to select bypass-preferred global loads and then apply fine control in runtime to pick thread blocks for which cache bypassing is applied. APCM dynamically determines the locality type of a global load by tracking access history at runtime. Compiler-based methods may predict locality types using load dependency graph, however real data sharing can be detected more accurately in runtime.

Hardware based GPU cache management methods focusing on cache bypass were also proposed. Chen et al. [9] suggested the cache line protection policy based on sampled reuse distances (PD) and the warp throttling method to avoid contention in cache and interconnection network. In order to decide the number of optimal active warps, they use the degree of cache contention by detecting victim cache lines and latency of interconnection network as resource congestion factors. Tian et al. [37] proposed the GPU cache management to bypass unnecessary memory requests predicted by the PC-based bypass predictor. Li et al. [25] presented the locality-driven cache bypassing method to exploit larger decoupled tag area to track locality information and predict requests to be bypassed. We compared our approach against the static cache line protection and the PC-based cache bypass schemes to show our approach provides higher performance.

GPU Warp Scheduling and Throttling: In GPU larger number of concurrent warps is normally expected to increase overall performance with better thread-level parallelism. However, it may increase cache contention. Thus several warp scheduling or throttling methods were suggested to increase efficiency of GPU cache. Kayıran et al. [19] proposed the algorithm that determines the optimal number of concurrent thread blocks to harmonize core utilization and contention of memory subsystem. Rogers et al. [33] revealed limiting the number of active warps can increase overall performance of cache sensitive applications by increasing cache hit ratio. They proposed the warp throttling algorithm that constrains the number of active warps based on cache thrashing. Rogers et al. [34] also proposed the warp throttling method increasing cache efficiency for divergent loads in a loop. Li et al. [26] tackle the underutilization of memory subsystem by reduced thread level parallelism. They classify warps as throttled warps, normal warps accessing cache, non-polluting warps bypassing cache to utilize resources of the interconnection network and external DRAMs. Oh et al. [31] presented locality-aware warp scheduler which reorders warp priority to reduce cache miss ratios. These approaches are warp-level approaches to alleviate congestion in cache, interconnection and memory channels. Our APCM exploits per-load cache management schemes to maximize cache utilization by enabling more fine-grained control. Our study shows the per-load cache control of APCM gains more performance benefit than the per-warp cache management.

CPU Cache Management: In CPU domains cache management is a well-researched area. Cache bypassing is an approach to reduce wasted cache lines by unnecessary blocks. Normally, cache bypassing schemes in CPU are applied to L2 cache or last level cache (LLC) since frequently reaccessed data is filtered in L1 cache. Kharbutli and Solihin [20] proposed counter-based cache management algorithm to bypass the dead data in L2 cache. In this algorithm dead blocks are predicted based on a prediction table and an event counter in each tag. Cache line protection is performed based on predicted life time of allocated data. Jaleel et al. [17] presented the cache replacement scheme based on predicted re-reference priority of data to preserve frequently reaccessed data from non-temporal data requests. Duong et al. [12] proposed the cache management policy using protecting distance (PD), where a cache line is protected for estimated number of accesses.

8 CONCLUSION

General purpose applications running on GPUs suffer significant memory bottlenecks. The massive thread level parallelism in fact causes frequent cache thrashing since the size of the cache per each warp is extremely small. Using detailed motivational analysis we reveal that global loads have various levels of sensitivity towards TLP. We categorize the global loads into four bins based on their cache access locality behavior. We then present locality-specific cache data management policy for each load to improve cache efficiency. We propose APCM a hardware based mechanism that automatically bins loads into different categories based on cache access history of individual loads in a predefined monitored warp. We make the observation that GPU applications exhibit strong access pattern similarity and hence it is possible to observe the access patterns of a single monitored warp to determine the load behavior across the entire application. APCM uses information gathered from the monitored warp to determine whether a load instruction exhibits little temporal reuse, in which case that load instruction is marked for bypassing the cache. For a load instruction that exhibits significant temporal reuse APCM protects the data fetched by the load instruction until the data reuse is complete. APCM gains 34% performance improvement for cache sensitive applications and on average 22% of improvement for all types of applications. Our evaluation also reveals that load-specific cache management approaches are very effective even for the cache moderate applications that exhibit a mix of data locality types. Combined with previously proposed warp throttling methods APCM significantly outperforms the state-of-the-art warp-based cache management schemes.

ACKNOWLEDGMENT

This work was supported by DARPA-PERFECT-HR0011-12-2-0020, NSF-CAREER-0954211 and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2015R1A2A2A01008281).

REFERENCES

[1] *FreePDK process design kit*. http://www.eda.ncsu.edu/wiki/FreePDK
[2] *GPGPU-sim manual*. http://gpgpu-sim.org/manual
[3] Mohammad Abdel-Majeed, Hyeran Jeon, Alireza Shafaei, Massoud Pedram, and Murali Annavaram. 2017. Pilot Register File: Energy Efficient Partitioned Register File for GPUs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA '17)*.
[4] AMD. *AMD Graphics Cores Next (GCN) Architecture*.
[5] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[6] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '12)*. IEEE Computer Society, Washington, DC, USA, 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*. IEEE Computer Society, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[8] Shuai Che and Kevin Skadron. 2014. BenchFriend: Correlating the Performance of GPU Benchmarks. *Int. J. High Perform. Comput. Appl.* 28, 2 (May 2014), 238–250. https://doi.org/10.1177/1094342013507960

[9] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 343–355. https://doi.org/10.1109/MICRO.2014.11

[10] Hyojin Choi, Jaewoo Ahn, and Wonyong Sung. 2012. Reducing Off-chip Memory Traffic by Selective Cache Management Scheme in GPGPUs. In *Proceedings of the Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*. ACM, New York, NY, USA, 110–119. https://doi.org/10.1145/2159430.2159443

[11] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702

[12] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 389–400. https://doi.org/10.1109/MICRO.2012.43

[13] Kayvon Fatahalian and Mike Houston. 2008. A Closer Look at GPUs. *Communication of the ACM* 51, 10 (Oct. 2008), 50–57. https://doi.org/10.1145/1400181.1400197

[14] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*. IEEE Computer Society, Washington, DC, USA, 407–420. https://doi.org/10.1109/MICRO.2007.12

[15] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 260–269. https://doi.org/10.1145/1454115.1454152

[16] Hynix. *1Gb GDDR5 SGRAM H5GQ1H24AFR Specification*. http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf

[17] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971

[18] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *Proceedings of the ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 15–24. https://doi.org/10.1145/2304576.2304582

[19] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 157–166. http://dl.acm.org/citation.cfm?id=2523721.2523745

[20] Mazen Kharbutli and Yan Solihin. 2008. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Trans. Comput.* 57, 4 (April 2008), 433–447. https://doi.org/10.1109/TC.2007.70816

[21] Gunjae Koo, Hyeran Jeon, and Murali Annavaram. 2015. Revealing Critical Loads and Hidden Data Locality in GPGPU Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '15)*. IEEE Computer Society, Washington, DC, USA, 120–129. https://doi.org/10.1109/IISWC.2015.23

[22] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*. IEEE Computer Society, Washington, DC, USA, 213–224. https://doi.org/10.1109/MICRO.2010.44

[23] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *IEEE International Symposium on High Performance Computer Architecture (HPCA '14)*. 260–271. https://doi.org/10.1109/HPCA.2014.6835937

[24] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[25] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 67–77. https://doi.org/10.1145/2751205.2751237

[26] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Keckler. 2015. Priority-Based Cache Allocation in Throughput Processors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA '15)*. 89–100. https://doi.org/10.1109/HPCA.2015.7056024

[27] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report. HP Laboratories.

[28] NVIDIA. *NVIDIA CUDA C Programming Guide*.

[29] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*.

[30] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*.

[31] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. 2016. APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 191–203. https://doi.org/10.1109/ISCA.2016.26

[32] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 86–98. https://doi.org/10.1145/2540708.2540717

[33] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 72–83. https://doi.org/10.1109/MICRO.2012.16

[34] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware Warp Scheduling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2540708.2540718

[35] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 73–82. http://dl.acm.org/citation.cfm?id=2523721.2523735

[36] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwe. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign.

[37] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jimenez. 2015. Adaptive GPU Cache Bypassing Categories and Subject Descriptors. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU-8)*. 36–47.

[38] Steven J. E. Wilton and Norman P. Jouppi. 1996. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits* 31, 5 (May 1996), 677–688.

[39] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 516–523. http://dl.acm.org/citation.cfm?id=2561828.2561929

[40] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA '15)*. 76–88. https://doi.org/10.1109/HPCA.2015.7056023

[41] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. 2014. Graph processing on GPUs: Where are the bottlenecks?. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '14)*. 140–149. https://doi.org/10.1109/IISWC.2014.6983053

[42] George L. Yuan, Ali Bakhoda, and Tor M. Aamodt. 2009. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*. ACM, New York, NY, USA, 34–44. https://doi.org/10.1145/1669112.1669119

[43] Mohammed Zackriya V and Harish M. Kittur. 2016. Precharge-Free, Low-Power Content-Addressable Memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* PP, 99 (2016), 1–8. https://doi.org/10.1109/TVLSI.2016.2518219