iWarp: Cache Interference-Aware Warp Scheduling for General Purpose Graphic Processing Units

Shuwen Gao
Computer Architecture and Memory Systems Laboratory
Department of Electrical Engineering
The University of Texas at Dallas
Shu-wen.Gao@utdallas.edu

Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory
Department of Electrical Engineering
The University of Texas at Dallas
jung@utdallas.edu

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

The text of the paper begins here.

2. Background

In this section, we briefly explain the fundamental organization of our baseline GPU device. We then look into the nature of cache contention in a streaming multiprocessor environment, which in turn can introduce a significant performance degradation in diverse data-intensive workloads. Finally, we discuss the three categories of conventional and recently proposed warp scheduling policies that are implemented by the existing GPGPU designs to counter the stifling effect of cache contention.

2.1 Baseline GPGPU Architecture

A GPU device in typical consists of multiple *streaming multiprocessors* (SMs), which are also known as *single instruction multiple thread* (SIMT) computational units. Figures 1 illustrates our baseline architecture, which essentially resembles NVIDIA's Fermi GPU [12], and the corresponding micro-architecture of the SM, respectively. Specifically, our baseline GPU employs 15 streaming multiprocessors with 32 CUDA cores and memory partition of 256 byte width. As shown in Figure 1, each memory partition is composed by a shared L2 cache, global DRAM and the corresponding memory controller (MC), and is attached to the interconnect network as off-chip resources. In contrast, a private L1 data cache (L1D), a read-only texture cache, a constant cache, and a shared-memory exist within each SM as on-chip storage resources. Thanks to this scalable and programmable hardware processors, it is significantly easier to map existing data-intensive applications to the massive parallel computation model of a modern GPU device compared to the earlier generations of hardwired pipeline GPUs [8]. With a state-of-the-art GPU programming interface, such as CUDA [11] or OpenCL [9], an GPU application can comprise of one or more computation *kernels* [11], also known as grids [11], each of which contains multiple threads. In our baseline architecture, a kernel can contain up to 23,040 lightweight threads. In practice, modern GPUs batch together groups of such individual threads, and execute them on an SM in lock-step fashion with single instruction, multiple data (SIMD) pipeline. Once a target kernel is launched into the GPU device, the lightweight threads are assigned to each SM as clusters of threads termed as *cooperative thread arrays (CTAs)*. These CTAs execute the same kernel program instance but process different chunks of data. The corresponding thread blocks (in a CTA) are synchronized by barriers to communicate with each other through shared and global memory [8]. The number of CTAs assigned to each SM depends on the on-chip resources available on the SM (i.e., register files, shared memory and barriers that are needed for each CTA). Within the SM, threads are managed, scheduled and executed in groups of 32, called *warps* [8] (also known as wavefront in AMD GPU [14]). The multiple threads in warp(s) are managed by a warp scheduler. Such thread groups share the same instruction, but process different data, in parallel. For example, in our baseline, it employs two warp schedulers for each SM. At the issue stage, per cycle, the schedulers select two warps from a list of ready warps to be executed based on a specific scheduling policy that we will explain shortly.

2.2 Cache Contention

While it is common for CPUs to accommodate several kilobytes of L1D cache memory per thread, GPUs can employ only tens of bytes per thread [5]. GPU applications usually have a large number of threads, and as a result the per-thread cache capacity can be as low as a few dozens of bytes; which can in turn generate a high rate of capacity misses in the L1D cache. Consider Algorithm 1 as a simple code example (one kernel) of *ATAX* in Polybench [3] which is one of memory intensive applications. One can observe from this *ATAX* kernel implementation that, through the iteration, each thread requests one entire row data from *A*[], which is a 2048*2048 matrix saved in the global memory (DRAM). For 4B floats, the data requested by each thread will be 8KB (4B*2048 = 8KB), which is half the size (or $\frac{1}{6}$) of a private 16KB (or 48KB) L1D cache in our baseline GPU architecture. As a consequence, even for a single

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

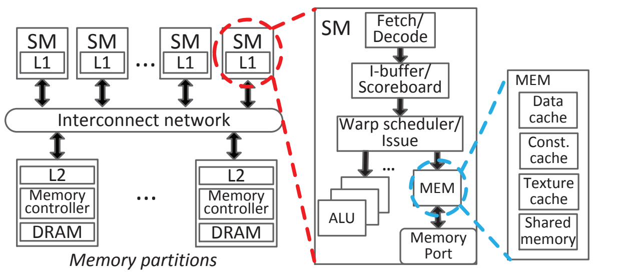


Figure 1. Baseline GPU architecture. While each streaming multiprocessor has its own warp scheduler and L1 cache as on-chip private resources, L2 caches are connected to the interconnection network as memory partitions which are shared by the multiple streaming multiprocessors.

warp (32 threads), the memory request can cause large number of capacity misses in the L1D cache, which cannot be solved solely by the warp scheduler.

Algorithm 1: atax_kernel1

```
//A[] is an 2048*2048 matrix, NY = 2048
//a tread block (one dimensional) contains 256 threads
int tread_id = blockIdx.x * blockDim.x + threadIdx.x
for $j \leftarrow 0$ to $NY$ do
    tmp[tread_id] += A[tread_id * NY + j] * x[j]
end
```

Since memory access behavior significantly varies based on the relationship of multiple warps' execution, we categorize the GPU cache contention as follows:

Inter-warp contention: This contention is triggered by the memory requests that are generated from different warps (either within the same CTA or coming from different CTAs) running on a target SM, but contest for the same cache set. Under many GPU workloads, large number of threads can be executed within the same target SM, and the subsequent memory accesses can easily exceed the L1 cache capacity, even for the relatively less memory intensive benchmarks. Several warp scheduling policies have been proposed, such as CCWS [13] and OWL [7], that aim to handle this kind of contentions in an attempt to improve the performance of L1D cache.

Intra-warp contention: Unlike the inter-warp contention, intra-warp contention is caused by the incoming memory requests that stem from the 32 threads within the same warp, and contest for the same cache set. For the memory intensive benchmarks, this kind of contentions unfortunately introduce many capacity misses in the L1D cache, and *cannot* be solved entirely by the warp scheduler [5]. This is because L1D cache in GPU is not able to maintain all the data requested even by a single warp due to its small storage capacity (e.g., 16KB - 48KB). In the previous *ATAX* example, the L1D cache can only maintain the data for two (with 16KB L1D cache) or six (with 48KB L1D cache) threads, and as the threads are executed in groups of 32 threads, many cache misses arise due to this intra-warp contention in L1D.

In addition, a warp with large data reuse is considered to have the potential *intra-warp locality* [13], which means the data in the L1 cache is initially referenced and re-referenced multiple times by the same warp had the L1 cache be large enough to accommodate all the requested data.

To show the significance of the L1D cache contention issue, we performed a simulation-based study using a cycle-level GPG-

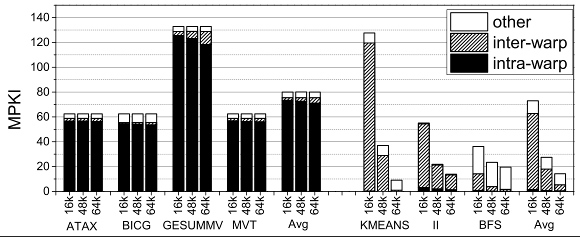


Figure 2. Misses per thousand instructions (MPKI) of L1D cache with different cache size. Cache misses are categorized into intra-warp contention, inter-warp contention and other (pure misses). For heavy memory intensive benchmarks, the misses mainly comes from intra-warp contention and changes very little with enlarged cache size. For moderate memory intensive benchmarks, the misses mainly comes from inter-warp contention and drops significantly with larger cache size.

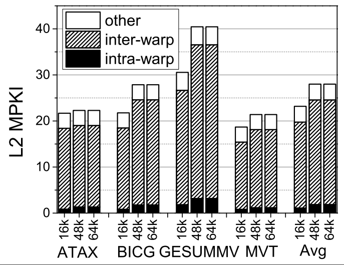


Figure 3. Misses per thousand instructions (MPKI) of L2 cache across memory intensive benchmarks with different L1D cache size. With a significantly larger size compared to L1D cache, the misses in L2 cache mainly comes from inter-warp contention.

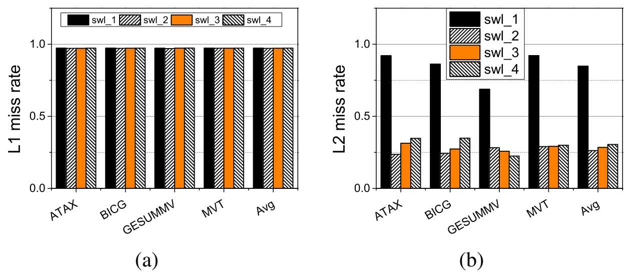


Figure 4. (a) L1D miss rate with SWL scheduler. With different number of active warps, the L1D miss rates are similar, because the cache size too small to accommodate the data for even a single warp. (b) L1D miss rate with SWL scheduler. The lowest miss rate is achieved when the number of active warps is less than the maximum except for GESUMMV, whose input size is smaller than other three benchmarks.

PU simulator [1]. In this work, we implement three different L1D cache sizes, i.e., the configurable cache size in Fermi architecture (16KB and 48KB) and a larger size of 64KB to show the potential performance. As shown in Figure 2, intra-warp contention can be responsible for majority of the misses in the L1D cache for several data intensive workloads (i.e., ATAX, BICG, GESUMMV and MVT[3], with scaled down input size [5]). Even with a larger 48KB or 64 KB L1D cache, intra-warp contention is still the main cause of the cache misses. One can observe from this figure that, with 16KB L1D cache, while misses per thousand instruction (MPKI) caused by inter-warp contention accounts for only 2.7% of total misses, 91.7% of total MPKI, on average, is caused by intra-warp contention. Even with a larger 64KB L1D cache, intra-warp contention is responsible for 88.8% of the total cache misses, while inter-warp contention only contributes 5.6%. Due to the insensitivity of contention to the L1D cache size, we classify this kind of benchmarks as *heavy memory intensive* benchmarks.

It should be noted that, for other workloads (i.e., BFS, KMEANS [2] and II [4]), even though majority of the cache misses are contributed by the inter-warp contentions with a small 16KB L1D cache, this challenge can be simply addressed if the L1D cache can be made larger. As one can see from Figure 2, once the cache size is enlarged to 48KB and 64 KB, the inter-warp contention is largely reduced by 71.8% and 92.1%, respectively. As the cache contention can be reduced significantly by utilizing larger size L1D cache, we categorize this kind of benchmarks as *moderate memory intensive* benchmarks.

Cache contention in L2: In contrast to the L1D GPU cache, the shared L2 cache is significantly larger (e.g., 768KB in Fermi architecture [12]), and therefore it has the potential to accommodate data requested by multiple warps. As in the data-intensive *ATAX* example, data requested by one entire warp is 256KB (2048*32*4), which is much larger than the L1D cache size and thus cannot be fully contained in L1D cache. However, it is less than the L2 cache size, and thus can be accommodated in L2 cache if there is not inter-warp contention at L2-level. As illustrated in Figure 3, the misses in L2 cache mainly comes from inter-warp contention. Since the L2-level inter-warp contention can come either from warps within the same SM or the warps across different SMs, we classify this kind of contention in L2 cache as *L2-level interference*. One can learn from this observation that even though the intra-warp contentions cannot be avoided in L1D cache, *the repeated data requests from the same warp can always hit on the shared L2 cache in the ideal case where the L2 cache can accommodate the data for a whole warp and there is no L2-level interference*. That is, although the L2 cache only have a few times more bytes per thread than the L1D cache dose when all SMs are fully utilized, once the number of active warps decreases, L2 cache has the potential to accommodate the entire data requested by fewer warps, and thus allocate more space for each thread.

As illustrated in Figure 1, the shared L2 cache sits behind the interconnection network, which results in a notably longer latency (hundreds of cycles [15]) than the private L1D cache accesses. Because of this, only sporadic attention has been paid to improve its performance. However, our study shows that, instead of directly improving the L1D cache performance, sacrificing the L2 cache latency to avoid even longer DRAM accesses can significantly improve the overall GPGPU performance in cases where cache misses cannot be avoided in L1D cache due to intra-warp contention issue.

2.3 Warp Scheduling

In order to hide the latency of the underlying memory accesses, a hardware-based warp scheduler selects/issues a ready warp from a warp list following a specific scheduling policy, and try to improve the computation throughput. We now look into the three categories of warp scheduling policies that are commonly used in modern GPU architectures.

REMARK [not implemented, should I?]??? Loose Round-Robin Scheduling (LRR): In this policy, multiple warps are ordered in the warp list based on their warp ID, and their executions are prioritized to be scheduled in a round-robin order. If the warp at the head of the list cannot be issued in the current cycle due to a long memory latency, it is logically moved to the end of the warp list, and the next warp in the list will get the highest priority. While LRR has the advantage of simple implementation, it does not consider the cache performance for most part, and thus can cause higher performance degradation compared to the other, more advanced schedulers; particularly when executing memory intensive applications.

Greedy-then-Oldest Scheduling (GTO): GTO priorities the oldest warp and runs it until it is stalled due to some long latency operation such as DRAM memory fetch. The age of a warp depends on the time it is assigned to the core. In this scheduling policy, if multiple warps have been assigned to a target core at the same time, a higher priority will be given to the one with the smaller thread IDs [13]. As GTO always gives the higher priority to older warps, it guarantees that such older warps have more chances to exclusively access the underlying memory, which in turn can exhibit better performance than LRR.

Cache-Locality-Aware Scheduling (CLA): Several cache-locality-aware warp scheduling policies have being proposed in recent years in order to alleviate the long memory latency, such as the Two-Level Warp Scheduling [10], Prefetch-aware Warp Scheduling [6], OWL scheduler [7] and Cache-conscious Wavefront Scheduling (CCWS) [13]. This type of dynamic warp scheduling strategies are designed for reducing the L1D cache misses for cache sensitive applications by determining how many and which warps should be executed based on the memory access behavior at runtime. Specifically, in this work, we implement CCWS as a comparison, which is devised based on the observation that the majority of the data reuse behavior behind the highly cache-sensitive benchmarks comes from the same warp, that is, those benchmarks exhibit L1-level intra-warp locality. Once such L1 level intra-warp locality is detected for certain warps, these warps will be given more exclusive access to the L1D cache by stalling the warps with less L1 level intra-warp locality. The goal here is to reduce the inter-warp contention, and increase the L1 cache hit by throttling the number of active warps to some extend.

It should be noted that, for benchmarks that exhibit memory requests of particularly large sizes, repeated data reuse from even a single warp can generate large L1D cache misses. Unfortunately, none of these three scheduling policies can effectively improve the L1D cache performance in such a scenario.

3. Interference-Aware Scheduling

3.1 Overview of iWarp

The cache-locality aware scheduling methods prioritize the warps experiencing high L1-level intra-warp locality in an attempt to reduce the "inter-warp contention" caused by many cache accesses of multiple active warps within a streaming multiprocessor. However, as shown in Figure 2 we observe that, for heavy memory intensive benchmarks, the majority of the misses in the L1D cache are generated by the repeated data requests from the same warps (i.e.,"intra-warp contention"). In other words, even with a single active warp, this intra-warp cache contention cannot be reduced effectively by the existing warp schedulers – we demonstrate this in Figure 4(a), where we apply static wavefront scheduling (SWL) [13] scheme, where the limit on the number of warps/wavefronts are specified by the programmer, to the heavy memory intensive benchmarks with

16KB L1D cache. As one can observe from Figure 4(a), even with the limit of only one active warp for each scheduler, the L1D cache miss rate is similar to when all warps are active (in this case, four warps for each scheduler). One of the insights behind this work is that, *such intra-warp contention problem can actually be addressed at the L2 level, if one can take advantage of the larger size of L2 cache to get a higher cache hit rate at L2-level*. That is, reducing the inter-warp contention shown in Figure 3 as much as possible to obtain a higher L2 hit rate, and thus avoid the long DRAM memory access. Motivated by this, we propose a cache interference-aware dynamic scheduling technique, iWarp, which can alleviate the L2-level interference and improve L2-level intra-warp locality. Even though L2 cache accesses exhibit relatively long latency, the data requested from the same warp can be served from the L2 cache, rather than having to retrieve it from the underlying global memory (DRAM), which in turn can save more than hundreds of cycles. In addition to hiding the long latency of accessing the DRAM, which can in turn improve the overall performance, our proposed iWarp can also increase the GPU's energy efficiency.

Further, the most conventional warp schedulers do not consider the cache interference between warps while choosing which warp is to be stalled. As a result, two warps that both exhibits L1-level intra-warp locality can contend for acquiring the same cache blocks, which can unfortunately generate many cache misses. In contrast, our proposed iWarp detects the cache contention between warp pairs, and stalls one of them to leave the other with more cache blocks in L2. In this way, iWarp can ensure that the data requested by the currently executing warp can reside in the L2 cache with fewer interference from other warps, and thus guarantee a high hit rate in the L2 cache.

Figures **??** and **??** illustrate a motivational example for our proposed scheme and the rationale behind it, respectively. Specifically, Figure **??** shows GPU performance for particularly data-intensive workloads using LRR, and varying L2 cache sizes. One can observe that, with an unlimited L2 cache size, those workloads have 86.7% performance improvement over the L2 cache configuration with a default GPU cache size (786 KB). Figure **??** demonstrates that, the reason behind this performance improvement is a sharp decline in L2 cache misses, which can be up to 97.0% compared to the baseline L2 cache size. Since, physically increasing the size of L2 cache can be expensive and hard to implement, our iWarp reduces the L2 cache misses by throttling the number of active warps that exhibit interference of many cache accesses which lead to significant cache thrashing. In this way, our iWarp can leverage the baseline L2 configuration without any architectural modification, and can approach the aspired behavior of an unlimited L2 cache.

3.2 Scheduling Strategy

3.2.1 L1-level

To make a better scheduling decision about eliminating the cache interference, our iWarp is capable of capturing the underlying cache contention by recording which warps send the memory requests. Once a pair of warps in the same SM is detected to have inter-warp contention between each other, our iWarp assigns the current executing warp priority to access L1 cache by stalling the other warp. In other words, for the current executing warp, our iWarp stalls the warp that causes the latest cache contention with it.

Figure 5 shows an example that illustrate the inter-warp contention caused by data reuse within a single SM in L1-level. For simplicity, we assume that there are four active warps, four sets in L1D cache and each set has one cache line. All cache lines are free at the initial state. W0(D0) means warp0 requests for data D0, and the arrow shows the data mapping to the cache line. As is shown by Figure 5 a), b) and c), the first three data requests do not cause any contention since the corresponding cache lines are free. When

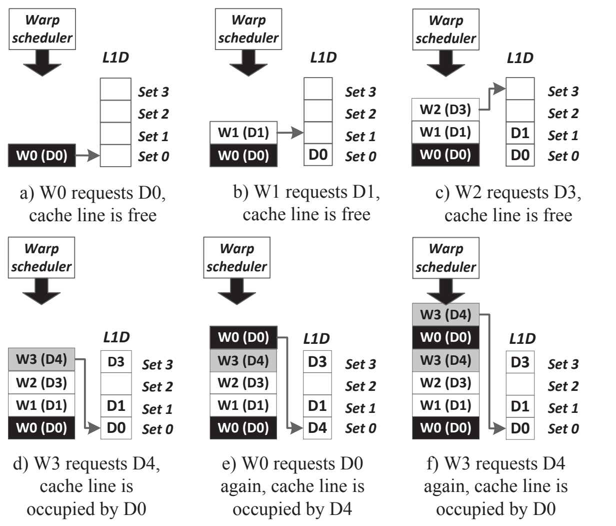

Warp scheduler
L1D
Set 3
Set 2
Set 1
Set 0
W0 (D0)
W1 (D1)
W2 (D3)
W3 (D4)
D0
D1
D3
D4
a) W0 requests D0, cache line is free
b) W1 requests D1, cache line is free
c) W2 requests D3, cache line is free
d) W3 requests D4, cache line is occupied by D0
e) W0 requests D0 again, cache line is occupied by D4
f) W3 requests D4 again, cache line is occupied by D0

Figure 5. Illustration of inter-warp contention in L1D cache. Data D0 and D4 that are rereferenced by warp W0 and W3, respectively, have inter-warp contention with each other.

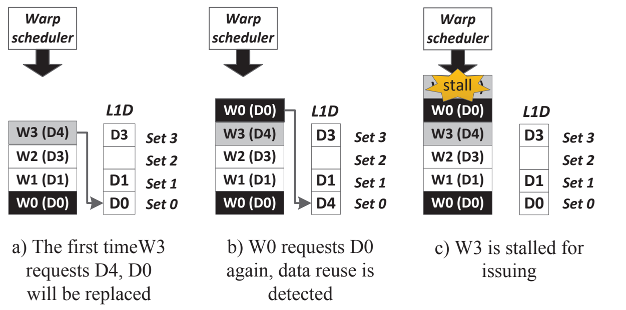

Warp scheduler
stall
L1D
Set 3
Set 2
Set 1
Set 0
W0 (D0)
W1 (D1)
W2 (D3)
W3 (D4)
D0
D1
D3
D4
a) The first timeW3 requests D4, D0 will be replaced
b) W0 requests D0 again, data reuse is detected
c) W3 is stalled for issuing

Figure 6. Illustration of iWarp working scheme on L1-level using the example in Figure 5. The first time warp W3 requests data D4, D0 in cache set0 will be replaced. Then when W0 requests D0 again, data reuse of D4 is detected, so as the contention between W0 and W3. iWarp stalls loading W3, and thus further data reuse of D0 from W0 will hit on L1D cache.

W3 requests for D4 in Figure 5 d), the corresponding cache line is occupied by D0 which was requested by W0. Therefore D0 will be replaced by the latest requested D4. When W0 is issued later on and requests for D0 again in Figure 5 e), conflict miss occurs since the cache line is occupied by D4. Similar situation happens when W3 is issued and requests for D4 again as is shown in Figure 5 f).

iWarp aims to detect the wrap pair that are contending for the same cache line (W0 and W3 in previous example in Figure 5), and stall one of them to leave the other more cache space so that the following data reuse of the remaining active warp can always hit on cache.

The modified cache structure that is able to detect the cache interference cause by data reuse is given in Section 4.1,

The following example illustrates the core strategy of our iWarp. Figure 6 a) corresponds to Figure 5 d) when W3 requests for D4 for the first time, and D0 will be replaced. In Figure 6 b), W0 requests for D0 again but causes cache miss since D0 has been replaced by D4. In such case, iWarp detects the data reuse of W0 and stops W3 for further issuing as is shown in In Figure 6 c). Therefore, further data reuse of D0 by W0 can hit on cache. In addition, when

W0 finishes executing or the cache contention. That is, when W0 does not request for data in cache set0 any more, W3 is reactivated, and its reuse of data D4 can hit on cache without the conflict from W0. In such way, iWarp reduces the L1 miss rate when the misses mainly come from inter-warp contention.

3.2.2 L2-level

In order to detect the L2-level interference, our iWarp needs to keep the history information for L2 cache. However, we propose to avoid such costly overhead by taking advantage of the large data reuse observed in GPU workloads, and detect the cache contention between warps from the smaller L1D cache instead. As a consequence, we can significantly reduce the hardware overhead for implementing i-Warp. The key idea here is that, the data that sits in L1D cache also occupies L2 cache blocks. In other words, the data that is fetched to L2 cache will eventually be fetched to L1D cache. As such, the L2-level interference can actually be detected in L1D cache as inter-warp contention, and thus no more further modification is needed. The following example illustrates the key idea of our iWarp at L2-level.

Figure 7 demonstrates a simplified example of the cache contention problem observed for data-intensive GPGPU workloads. In this example, we assume that there are three cores, and each core has two active warps for the sake of brevity. The L1D cache in each core can only hold data for one-fourth of threads in a warp (i.e., eight threads), while the L2 cache is able to maintain all the data for three warps. As a result, data of all six warps will contend for the limited L2 cache blocks. For L1D cache, as shown in Figure 7(a), when the requested data for one warp is fetched from L2 to L1D cache, the incoming data will cause frequent data replacements in L1D cache due to the data being larger than the cache capacity. For example, since data for warp0 in core0 (C0W0) is four times larger than the L1D cache size, newly incoming data will replace the data for the same warp that already occupies L1D cache blocks. Due to the large data reuse characteristic in the data-intensive workloads, such frequent data replacements in turn can introduce a high L1D miss rate.

Figure 7(a) represents an ideal case where, the entire data requested by the currently executing warp can be found in the L2 cache for each core. As a result, irrespective of the huge L1 misses caused by intra-warp contention, L2 fully catches the intra-warp locality and can generate cache (excluding the cold start) hits in most cases, which in turn can avoid the long latency of accessing the slow DRAM-based GPU memory.

Figure 7(b) illustrates the situation regarding L2 cache interference. Once the executing warp in core0 switches to warp1 due to some long latency operation of warp0, a portion of the blocks in L2 cache will be replaced by the newly requested data. Since the data for core1 and core2 is replaced, the future requests from these two cores will introduce L2 cache misses, which will result in time-consuming data extraction from the underlying DRAM. Further, once the data is fetched from the DRAM for core1 and core2, the replacement operation in L2 cache can aggravate the cache interference.

The detection of cache interference in our iWarp scheduler is triggered when core0 switches back to executing warp0 from Figure 7(b) to Figure 7(c). Because of the data reuse feature, warp0 will request for the same data as in its previous execution. In Figure 7(c), the block in core0 indicates the old data "C0W1" in L1D cache before it is replaced by the new incoming data "C0W0". Upon the data replacement, by figuring out that the current requested data "C0W0" *was* in L1D cache, iWarp can identify this cache miss as a result of inter-warp contention. That is, the L2-level interference eventually propagated to L1-level and shown as the form of inter-warp contention. Once warp1 in core0 that introduces the interference is stalled as shown in Figure 7(c), data "C0W1" will stop loading from DRAM to L2 cache. In this way, the frequent data replacement problem in L2 can be alleviated, and more L2 cache blocks can be allocated to the other executing warps, which can in turn reduce L2 misses.

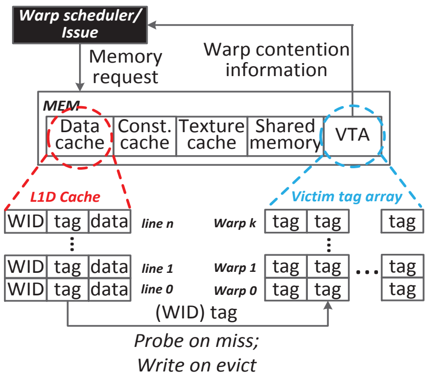


Figure 8. Modified cache structure for our proposed iWarp scheme. A warp ID (WID) is added to each L1D cache line and a victim tag array (VTA) is introduced for each warp. The warp contention information is detected in VTA upon cache miss, and is fed back to the warp scheduler.

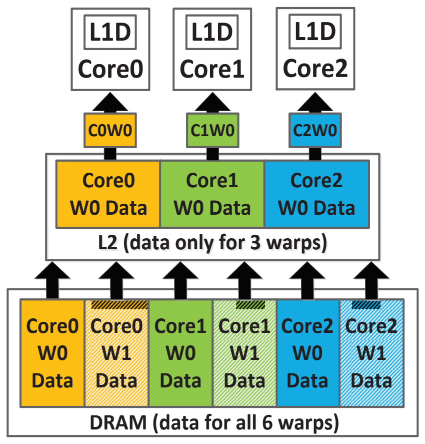
In real GPU workloads, as there are many more active warps and much larger data set, the cache interference can be significantly more severe, which requires intensive levels of warp scheduling for capturing L2-level intra-warp locality. In summary, even when the L1D cache misses cannot be avoided for data intensive workloads due to the small cache size, our iWarp aims to keep the requested data in L2 for a longer time by reducing the cache interference, so that the repeatedly requested data can hit on the L2 cache more frequently, improving overall performance.

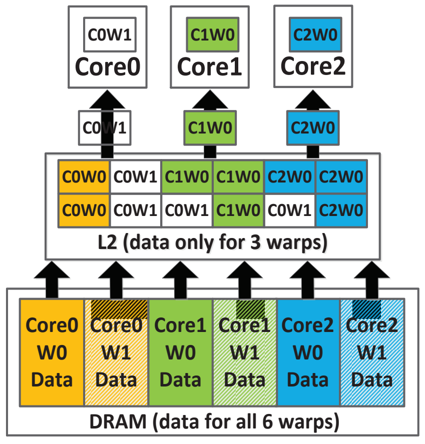
4. Scheduler Implementation

4.1 Cache Structure Modification

To detect the both the L1 and L2-level interference through L1-level cache contention, we use the modified cache structure as shown in Figure 8. A *warp ID (WID)* is added to each cache line of L1D cache in order to identify the warp reserving it. In addition, each warp is associated with a *victim tag array (VTA)* in the memory unit. When a cache line is reserved on a cache miss, the warp ID associated with the cache line is written into the WID field. On the other hand, if a cache line is evicted from the L1D cache, the corresponding tag of that line is written into the victim tag array associated with that warp. In this way, VTA keeps the record of each warp's memory request history. Whenever there is a miss on the L1D cache, the VTA associated with the warp that sent the memory request is probed. If the requested tag is found in the VTA, it means that the currently executing warp generates a same data request as the one found in its request history. If the current miss is due to an inter-warp contention, the underlying L2 cache interference is detected. In this case, though there is little we can do to address the L1-level intra-warp locality due to the limited L1D cache size for heavy memory intensive workloads, our iWarp will leave more cache blocks for this executing warp in the L2 cache to mitigate the L2-level interference, and thereby catch the intra-warp locality in L2-level; which in turn can avoid the longer DRAM memory access.

An illustration of how the modified cache is employed in the case of Figure 6 is shown in Figure 9. Figure 9(a) illustrates when the data $D0$ for warp W0 is going to be replaced by W3's data $D4$ corresponding to Figure 6 a). Before the replacement, tag $T0$


(a) Without L2-level interference.


(b) With L2-level interference.

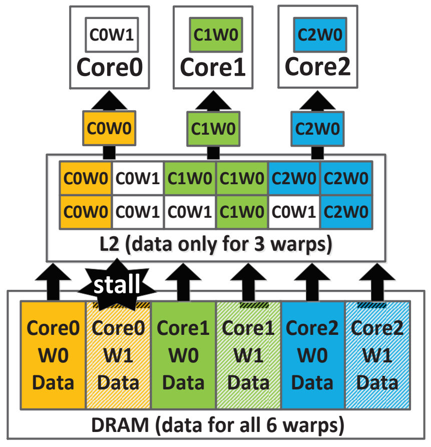
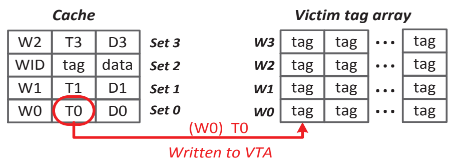

(c) L2-level interference eliminated.

Figure 7. A representative example of the cache contention problem and our iWarp scheduling strategies. (a) An ideal case where despite severe intra-warp contentions in L1D cache, there is no inter-warp contention in the L2 cache. Every data missed in the L1D cache can always hit in the L2 cache. (b) Newly requested data for warp1 in core0 (C0W1) occupies the cache blocks in L2, which introduces inter-warp contention in L2 cache. This contention causes L2 misses for other cores as their data has been replaced and can eventually be detected in core0. (c) Once the cache interference is detected, iWarp stalls the warp1 in core0, and the corresponding data will be stalled loading to L2 cache, which will reduce the L2 misses.


(a)

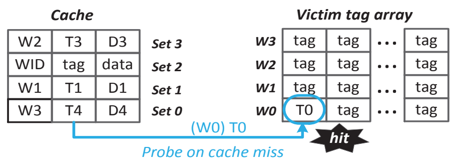

(b)

Figure 9. iWarp strategy details. (a) When a cache line is evicted from the L1D, its tag is written into the VTA portion that corresponds to the warp reserving it (marked using the WID). (b) Upon a cache miss, the requested data's tag is probed in the requesting warp's corresponding VTA. If the tag is found, a hit signal is generated.

is written to W0's portion of VTA. When W0 is issued and it requests for $D0$ again in Figure 6 b), W0's portion of VTA is probed, following the miss in L1 cache as shown in Figure 9(b). Since $T0$ exists in VTA, a hit signal is generated, which indicates that the currently requested data $D0$ *was* in the cache. And thus the inter-warp contention is detected.

4.2 Cache Interference Aware Scheduling

5. Evaluation

5.1 Methodology

5.2 Experimental Results

5.2.1 Performance Analysis

IPC. Figure 10
Cache Misses in L1 and L2. Figure 11 and Figure 12
DRAM accesses. Figure 13

No. of SMs	15
Warp size	32
No. of warp schedulers	2
No. of threads per SM	1536
L1D cache/Shared Memory per SM (i)	16KB, 128B line, 4-way assoc./48KB
L1D cache/Shared Memory per SM (ii)	48KB, 128B line, 6-way assoc./16KB
L1D cache/Shared Memory per SM (iii)	64KB, 128B line, 8-way assoc./16KB
Unified L2	786KB, 128B line, 8-way assoc.
L1D policy	allocate on miss, local write-back, global write-through
L2 policy	allocate on miss, write back

Table 1. GPGPU-Sim configuration.

Benchmarks	Description	Suite
	Heavy memory intensive	
ATAX	Matrix Transpose and Vector Multiplication	[3]
BICG	BiCG Sub Kernel of BiCGStab Linear Solver	[3]
GESUMMV	Scalar, Vector and Matrix Multiplication	[3]
MVT	Matrix Vector Product and Transpose	[3]
	Moderate memory intensive	
KMEANS	K-means Clustering	[2]
II	Inverted Index	[4]
BFS	Breadth First Search	[2]

Table 2. GPGPU benchmarks.

A. Appendix Title

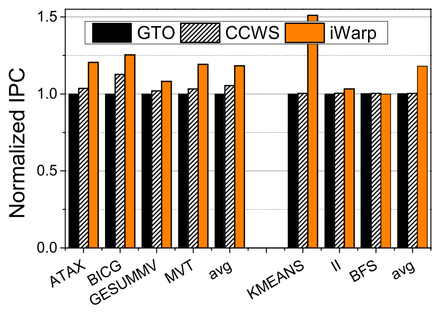
This is the text of the appendix, if you need one.

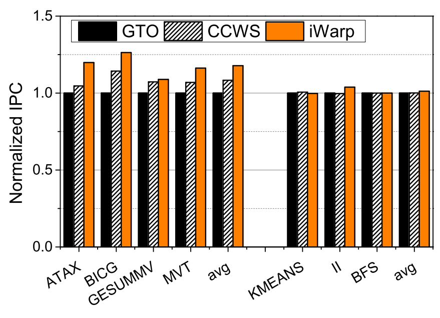
Acknowledgments

Acknowledgments, if needed.

References

[1] Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., and


(a) 16KB.


(b) 48KB.

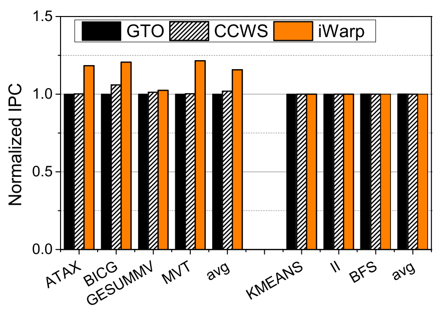
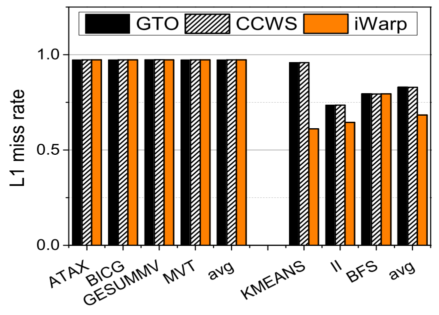
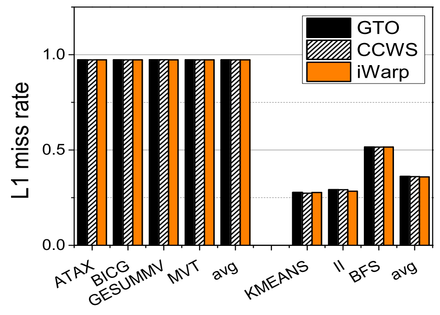

(c) 64KB.

Figure 10. Overall performance improvement of CCWS and iWarp with different size of L1D cache, normalized to GTO. With a small 16KB L1D cache, CCWS and iWarp improve the performance of heavy memory intensive benchmarks of 5.4% and 18.3% over GTO, respectively. With a larger 48KB L1D cache size, CCWS and iWarp improve the performance of heavy memory intensive benchmarks of 8.3% and 17.8% over GTO, respectively. As the cache size is enlarged to 64KB, the improvement of CCWS decreases to 1.9%, while iWarp sill achieves 15.8% improvement over GTO. As the L1D cache size becomes larger, cache interference of moderate cache sensitive benchmarks decreases significantly, and thus CCWS and iWrap tend to have similar performance with GTO.


(a) 16KB.


(b) 48KB.

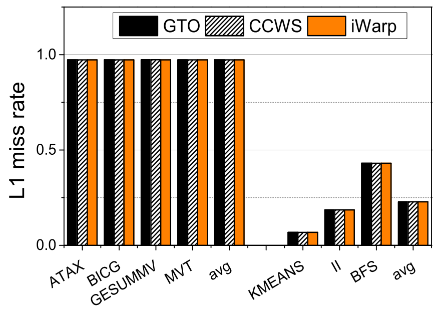
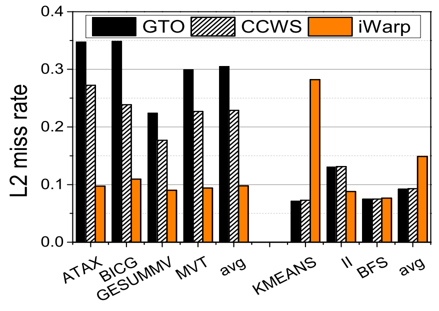
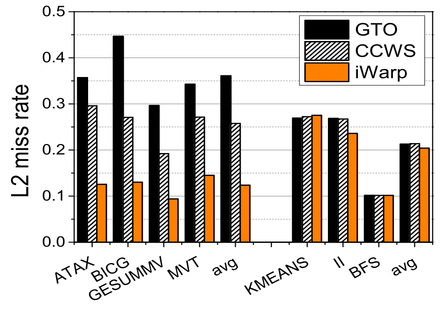

(c) 64KB.

Figure 11. L1 miss rate of three schedulers. For heavy memory intensive benchmarks, the L1 miss rates with different warp schedulers are similar due to the huge intra-warp contention. For moderate memory intensive benchmarks, CCWS and iWrap reduces the L1 miss rate with small 16KB L1D cache, and have similar L1 miss rate with GTO with larger L1D cache.


(a) 16KB.


(b) 48KB.

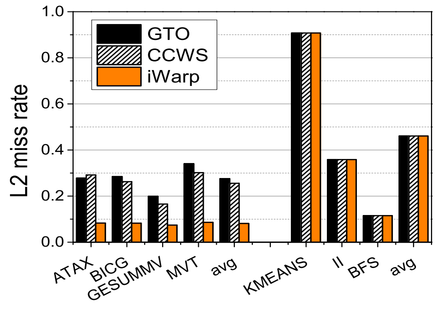
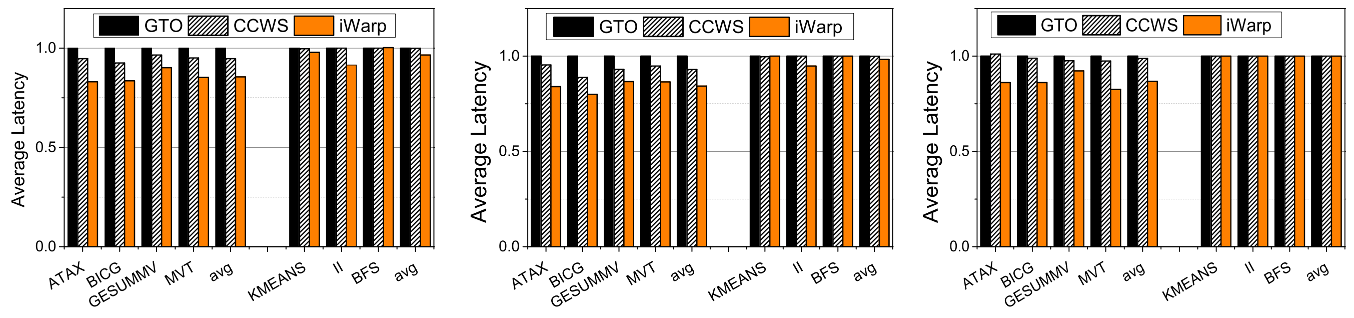

(c) 64KB.

Figure 12. L2 miss rate of three schedulers. For heavy memory intensive benchmarks, iWarp achieves the lowest L2 miss rate among the three schedulers with different size of L1D cache.



(a) 16KB. (b) 48KB. (c) 64KB.

Figure 13. Average memory fetch latency of three schedulers, normalized to GTO. For heavy memory intensive benchmarks, iWarp achieves the shortest average memory fetch latency among the three schedulers with different size of L1D cache.

AAMODT, T. M. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), IEEE, pp. 163–174.

[2] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), IEEE, pp. 44–54.

[3] GRAUER-GRAY, S., XU, L., SEARLES, R., AYALASOMAYAJULA, S., AND CAVAZOS, J. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012* (2012), IEEE, pp. 1–10.

[4] HE, B., FANG, W., LUO, Q., GOVINDARAJU, N. K., AND WANG, T. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (2008), ACM, pp. 260–269.

[5] JIA, W., SHAW, K. A., AND MARTONOSI, M. Mrpb: Memory request prioritization for massively parallel processors. In *20th International Symposium on High Performance Computer Architecture (HPCA-20)* (2014).

[6] JOG, A., ET AL. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ACM, pp. 332–343.

[7] JOG, A., ET AL. Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance. *ACM SIGARCH Computer Architecture News 41*, 1 (2013), 395–406.

[8] LINDHOLM, E., ET AL. Nvidia tesla: A unified graphics and computing architecture. *Ieee Micro 28*, 2 (2008), 39–55.

[9] MUNSHI, A., ET AL. The opencl specification. *Khronos OpenCL Working Group 1* (2009), l1–15.

[10] NARASIMAN, V., ET AL. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 308–317.

[11] NVIDIA, C. Programming guide, 2008.

[12] NVIDIA, C. Nvidia's next generation cuda compute architecture: Fermi. *Computer system 26* (2009), 63–72.

[13] ROGERS, T. G., O'CONNOR, M., AND AAMODT, T. M. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), IEEE Computer Society, pp. 72–83.

[14] SDK, A. S. v2. 01 performance and optimization. *Technical Notes. Advanced Micro Devices Inc.: Sunnyvale, CA* (2010).

[15] WONG, H., PAPADOPOULOU, M.-M., SADOOGHI-ALVANDI, M., AND MOSHOVOS, A. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 235–246.