2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs

Yunho Oh*, Keunsoo Kim*, Myung Kuk Yoon*, Jong Hyun Park*, Yongjun Park†,
Won Woo Ro*, and Murali Annavaram‡
* *Yonsei University*
{*yunho.oh, keunsoo.kim, myungkuk.yoon, park.jonghyun, wro*}*@yonsei.ac.kr*
† *Hongik University*
yongjun.park@hongik.ac.kr
‡ *University of Southern California*
annavara@usc.edu

***Abstract*—Long memory latency and limited throughput become performance bottlenecks of GPGPU applications. The latency takes hundreds of cycles which is difficult to be hidden by simply interleaving tens of warp execution. While cache hierarchy helps to reduce memory system pressure, massive Thread-Level Parallelism (TLP) often causes excessive cache contention. This paper proposes Adaptive PREfetching and Scheduling (APRES) to improve GPU cache efficiency. APRES relies on the following observations. First, certain static load instructions tend to generate memory addresses having very high locality. Second, although loads have no locality, the access addresses still can show highly strided access pattern. Third, the locality behavior tends to be consistent regardless of warp ID.**

APRES schedules warps so that as many cache hits generated as possible before any cache misses generated. This is to minimize cache thrashing when many warps are contending for a cache line. However, to realize this operation, it is required to predict which warp will hit the cache in the near future. Without directly predicting future cache hit/miss for each warp, APRES creates a group of warps that will execute the same load instruction in the near future. Based on the third observation, we expect the locality behavior is consistent over all warps in the group. If the first executed warp in the group hits the cache, then the load is considered as a high locality type, and APRES prioritizes all warps in the group. Group prioritization leads to consecutive cache hits, because the grouped warps are likely to access the same cache line. If the first warp missed the cache, then the load is considered as a strided type, and APRES generates prefetch requests for the other warps in the group. After that, APRES prioritizes prefetch targeted warps so that the demand requests are merged to Miss Status Holding Register (MSHR) or prefetched lines can be accessed. On memory-intensive applications, APRES achieves 31.7% performance improvement compared to the baseline GPU and 7.2% additional speedup compared to the best combination of existing warp scheduling and prefetching methods.

***Keywords*-GPGPU; Warp Scheduling; Data Prefetching;**

I. Introduction

Modern Graphics Processing Units (GPUs) exploit Thread-Level Parallelism (TLP) to achieve high throughput. However, the performance of GPUs is often limited by memory system performance [1], [2], [3], [4], [5], [6]. In particular, the long off-chip memory access latency is a key limiter. The minimum access latency to off-chip DRAM takes hundreds of cycles. Even though GPUs support wide memory access width to support accesses from multiple threads, the per thread bandwidth is still a performance bottleneck. It has been shown that limited memory bandwidth often adds long queuing delay [1], [7]. While GPUs try to hide some of access latency by interleaving warp execution, the long off-chip memory delays are still exposed and the pipeline is stalled waiting for response from the memory system.

Significant efforts have been made for hiding effective access latency by improving cache efficiency [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. These techniques increase cache hits by throttling the number of active threads to reduce cache contention [14], or improving thread scheduling to maximize cache hits [15]. While these techniques are effective, the L1 cache hit rates are still very low even with these approaches due to the small L1 cache size. For instance, a typical size of GPU L1 cache is in the range of 16-48 KB per Streaming Multiprocessor (SM), which is one core within a GPU. However, the number of active threads per SM is in the range of 1024-2048. As a result, each thread, on average, has access to only 8-48 bytes of L1 cache [18]. This capacity is insufficient and often causes cache thrashing [19]. In our simulation studies (details later), capacity and conflict misses account for 42.7% of total misses.

Since GPU applications often show regular and predictable data access pattern, data prefetching techniques on GPUs have been proposed [20], [21] to reduce effective memory latency. However, when the per thread cache size is so severely restricted as in GPUs, prefetching has to achieve extremely high accuracy and near-perfect timeliness. Otherwise prefetched lines may evict useful data, or prefetched lines themselves may be evicted before they are used. These evictions increase memory traffic and queuing delay, resulting in unintended negative consequences.

This paper tackles the above challenges by exploiting two important observations that were made through detailed per load instruction characterization across a range of GPU benchmarks. Load instructions can be divided into the following two types. The first type of loads access only a small range of memory space, thus exhibit strong locality. An extreme case example is when a load is used for accessing a variable shared across all warps that load always accesses

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.26

the same address. The second type of loads have a large memory footprint and their data is never reused after the first access, however addresses from these loads still exhibit strong striding pattern. These loads are often used to access a large array indexed by thread ID, which is a common pattern observed in many GPU applications. The characteristics are unique to each static load regardless of warp IDs.

Using the above observation, this paper proposes a technique to improve cache efficiency, namely *Adaptive PREfetching and Scheduling (APRES)*. Since each static load is either striding or has high locality independent of the warp, APRES first groups warps that are predicted to execute the same load instruction in the near future. Once any warp in the group hits in the cache, APRES considers that load to exhibit high locality and predicts all other warps in the group will also hit the cache. As such, APRES prioritizes all warps of that group as they are predicted to hit in the cache. On the contrary, if a warp load misses in the cache, APRES characterizes that load as a stride type. APRES then issues prefetch requests targeted for all other warps that belong to that group. Given these warps exhibit stride behavior, it is possible to predict future demand requests generated from other warps based on the memory accesses executed by the current warp. Once the prefetch requests are issued, APRES prioritizes warps in the group to quickly use the prefetched data. This prioritization minimizes the possibility of early eviction of the prefetched lines. If the warps targeted for prefetch issue the load before the prefetched data is delivered, the demand requests are merged in miss status handling registers (MSHRs) of the L1 cache.

Detailed simulation results show that APRES outperforms the baseline GPU by 31.7% and it outperforms the best combination of existing warp scheduling and prefetching techniques by 7.2% on memory-intensive applications. In addition, APRES reduces the energy consumption by 10.8% compared to the baseline GPU.

The main contributions of this paper are as follows.

- We provide a detailed cache access characterization of static loads in GPU applications. We categorize the load instructions as either striding or exhibiting strong locality. We observe that static loads exhibit the same behavior across different warps in the same kernel.
- We propose APRES, a technique that improves cache efficiency by combining warp scheduling and prefetching. APRES groups and prioritizes a group of warps according to load instruction characteristics. Group-based scheduling improves cache efficiency by reducing cache contention and minimizing early evictions of prefetched cache lines.

The rest of the paper is organized as follows. Section II introduces the baseline GPU architecture. Section III describes the challenges associated with warp scheduling and prefetching. Section IV discusses the details of the APRES architecture. The experimental results are shown in Section V. Section VI explains the related work and we conclude in Section VII.

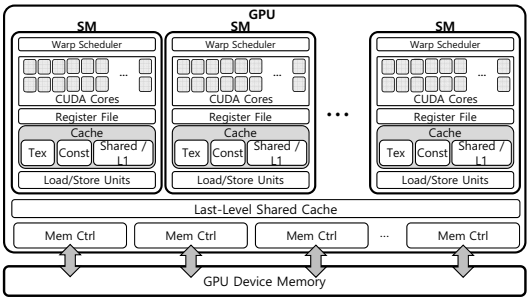


Figure 1. Baseline GPU architecture

II. Baseline Architecture

Figure 1 depicts the baseline GPU architecture used in this paper, which is based on NVIDIA's publicly disclosed architecture specifications. A GPU contains hundreds or thousands of cores. They are arranged in a hierarchical structure. A single GPU consists of multiple Streaming Multiprocessors (SMs). Each SM has dozens of cores (in NVIDIA GPU model, they are called CUDA[1] cores) which are in charge of arithmetic operations. In case of the Maxwell architecture, an SM contains 128 CUDA cores [22]. The GPU performs massive parallel processing based on the *Single Instruction Multiple Thread (SIMT)* execution model [23]. To efficiently execute threads on GPUs, they are scheduled based on an unique unit of thread group, called a warp. Each warp consists of 32 threads in case of NVIDIA GPUs. All threads in a warp execute the same instruction concurrently. Each SM stores the architecture state of dozens of warps, and the warp scheduler in an SM find the available warps from this pool of warps to be issued for execution every cycle. The basic warp scheduling scheme is Loose Round-Robin (LRR) policy. The warp scheduler with LRR policy provides equal scheduling priorities to all ready warps and finds an issuable warp in sequential order of warp IDs.

GPUs have a hierarchical memory system. Each SM has register file, shared memory, and L1 cache. Register file stores per-thread data generated from computations. Shared memory is used as a scratchpad space and can be configured by programmer. L1 data cache is transparent to the GPU device memory (also called global memory) [24]. L1 texture and constant cache stores read only data. Texture cache is useful in graphics processing. In the lower level, the last-level cache is shared by all SMs. Last-level cache is partitioned and each LLC partition is dedicated to each DRAM partition. Outside of the GPUs, the off-chip device memory is located as the lowest level of memory hierarchy.

To support massive memory operations efficiently, GPUs provide memory request coalescing. The memory requests are coalesced if threads in a warp access consecutive addresses in the device memory. In that case, the memory requests are unified, so that the traffics can be dramatically reduced [23].

[1]CUDA is the abbreviation of *Compute Unified Device Architecture*.

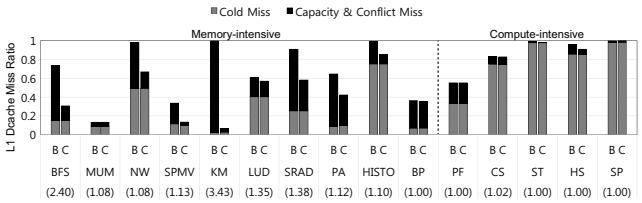


Figure 2. Breakdown of total L1 data cache miss ratio, capacity and conflict miss ratio of Baseline 32KB (B) and large 32MB Capacity of L1 data cache (C) (Number in parentheses: Relative performance of C normalized to B)

III. Understanding GPU Cache Misses

A. Impact of Memory System on Performance

L1 data cache on GPUs often suffers from high miss rate due to its small size. We measured the fraction of capacity, conflict, and cold misses for 15 applications. The detailed methodology is presented in Section V. We grouped capacity and conflict misses together because both misses happen due to limited cache size. A cache access request is considered either a capacity or a conflict miss if the line has been loaded to cache previously but evicted prior to first reuse. The Y axis of Figure 2 indicates L1 cache miss rate and each stacked bar indicates the portion of cold and capacity+conflict miss. We categorize the applications broadly into compute and memory-intensive according to their speedup sensitivity in memory system performance, such as the impact of increasing cache size or memory bandwidth. Although the miss rate of compute-intensive applications (87.6%) is higher than memory-intensive (67.1%), when the miss rate is reduced the benefit will be much more in memory-intensive applications. Capacity and conflict misses are the dominating misses in memory-intensive applications, which account for 62.8% of the miss rate.

As capacity or conflict misses account for significant portion of total cache misses, a larger cache size will improve performance. We will use a hypothetical GPU model for analyzing performance impact of memory system. The model uses a very large L1 cache of 32 MB. As shown in Figure 2, the large cache cuts the capacity and conflict misses into half in BFS, SPMV, and KM, compared to the baseline. The number in the parenthesis below the benchmark label on the X-axis shows the performance improvement (reduction in execution time) with the larger cache compared to the baseline. Based on these results, we find that reducing the capacity and conflict misses would result in significant performance improvements.

B. Characterizing Load Instructions of GPU Applications

In this section we characterize the cache access behavior of individual static loads in each of the GPU applications. Table I shows the list of top three frequently executed static load instructions for each application. For this study, we considered only the most memory-intensive kernel of each application. We measure the number of memory references

Table I
Characteristics of Frequently Executed Loads

App	PC	%Load	#L/#R	Miss Rate	Stride	%Stride
BFS	0x110	51.6%	0.04	0.78	0	16.3%
	0xF0	26.4%	0.12	0.90	0	13.3%
	0x198	9.5%	0.11	0.83	0	14.7%
MUM	0x7A8	66.2%	0.01	0.17	0	36.3%
	0x460	21.3%	0.04	0.04	0	46.8%
	0x8A0	12.3%	0.07	0.17	0	34.3%
NW	0x490	18.9%	0.98	1.0	-1966080	56.0%
	0xD18	18.8%	0.97	1.0	-1966080	74.5%
	0x108	1.8%	0.94	1.0	-1966080	60.8%
SPMV	0x1E0	51.5%	0.13	0.32	0	24.0%
	0x200	23.8%	0.25	0.25	0	19.3%
	0xE0	7.2%	0.65	0.81	0	12.5%
KM	0xE8	100.0%	0.03	0.99	4352	78.2%
LUD	0x20F0	30.2%	0.58	0.96	2048	66.6%
	0x2080	30.2%	0.57	0.91	2048	83.3%
	0x22E0	30.1%	0.66	0.97	2048	77.3%
SRAD	0x250	31.2%	0.99	0.99	16384	78.2%
	0x230	31.2%	0.99	1.0	16384	75.0%
	0x350	31.2%	0.52	0.99	16384	80.7%
PA	0x2210	51.7%	0.03	0.98	8832	42.7%
	0x2230	39.9%	0.002	0.16	0	36.2%
	0x2088	3.2%	0.02	0.02	256	91.5%
HISTO	0x168	100.0%	1	1.0	512	20.8%
BP	0x3F8	19.4%	0.59	1.0	128	75.5%
	0x408	19.4%	0.59	1.0	128	64.1%
	0x478	19.4%	0.59	0.03	128	67.1%

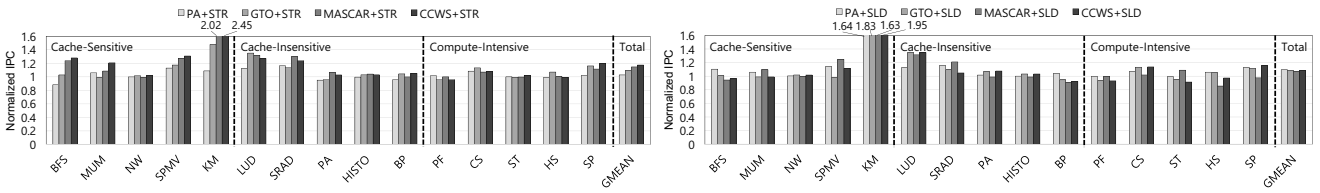
%Load: Portion of each load among total load executions
#L/#R: # of unique cache lines per reference, **Miss Rate**: L1 data cache miss rate
Stride: Stride in bytes, **%Stride**: Portion of stride among total stride detected

generated after coalescing multiple thread requests from a single warp as much as possible.

The column %Load indicates the fraction of memory requests generated by each static load over the total number of memory requests. #L/#R indicates the ratio between the number of unique cache lines and the number of cache references generated by the static load. With this metric, we can estimate how much memory locality exists between warps executing the same static load. If data referenced by a warp is re-referenced by the same warp or another warp [14] then #L/#R value will be small. Hence, in principle the load should have very high cache hit ratio.

We compare the #L/#R value with the actual cache miss rate suffered by that particular load instruction. As cache contention increases, the gap between #L/#R and miss rate increases. An extreme case is observed in the KM application. In KM, 100% of memory request are generated from only one static load. That load has #L/#R=0.03, indicating that the miss rate of 0.03 can be achieved with infinite cache size. However, the actual cache miss rate is almost 99%. Such a large gap between #L/#R and miss rate implies that severe cache thrashing made L1 cache virtually useless. In fact, we observed that the working set size of KM is about 2 MB per SM, which is more than 60× the size of L1 cache, which is 32 KB.

A simple solution is using a large L1 cache at the cost of higher storage and design overheads. As shown in Figure 2, using cache as large as 32 MB improves L1 cache hit rate significantly and hence the highest speedup of 3.4 is observed for KM. Clearly, such a large cache design is impractical using current design technologies. Another solution



(a) STR Prefetcher

(b) SLD Prefetcher

Figure 3. Relative performance comparison of prefetching schemes (Normalized to baseline)

is to dynamically control the number of active warps for mitigating excessive cache contention [14]. Warp throttling techniques are effective in mitigating cache contention but still such techniques do not fully consider characteristics of each static load. For instance, first two loads (0x250, 0x230) of SRAD have #L/#R value of around 1, meaning all memory references demand different cache lines. Hence, it is clear that those two loads have very high miss rate of 99%. Meanwhile, in the case of the third load (0x350) #L/#R of 0.52, which implies only half of the cache accesses request unique cache lines; however, similar to the first two loads the third load also shows miss rate of 99%. We surmise that cache lines accessed by the third load are evicted by the other two loads, which reduces opportunity improving cache hit rate of the third load. In Section V, we will provide detailed evaluation using warp throttling techniques.

In addition, Table I presents stride characterization of each of the static loads. For a pair of accesses, the stride is calculated by dividing the difference of the two addresses by the difference of the warp IDs. An ID of a warp is defined as the index of the first thread divided by warp size (32). We show the stride values only for the most frequently observed stride for each load. The *Stride* column of Table I shows the most common stride value and %Stride column indicates the fraction of the most frequent stride over the accesses generated by that static load.

While some static loads exhibit low memory locality, they show a very strong stride behavior. For instance, the first two loads of SRAD have high #L/#R value implying no memory locality, however the loads have strong stride behavior. 78.2% and 75.0% of accesses from the two loads have the regular stride value of 16,384.

C. GPU Prefetching Opportunities

Strong stride behavior enables prediction of future addresses, which makes prefetching a desirable option [25], [26], [27], [28], [29], [30]. In particular, when the application has neither temporal nor spatial locality, L1 data cache can be repurposed as a buffer for storing prefetched cache lines.

We analyze two state-of-the-art GPU prefetching methods. STRide prefetching (STR) [20], [21] uses a table in which each entry stores the PC address of each static load instruction and a base memory address and stride value. The stride calculation is done for each load instruction. Spatial Locality Detection based prefetching (SLD) [10] is the macro block-based prefetching method. A macro block consists of consecutive four cache lines. If two lines of the block are accessed, the SLD prefetcher will automatically generate prefetch requests for the remaining two lines in the same macro block.

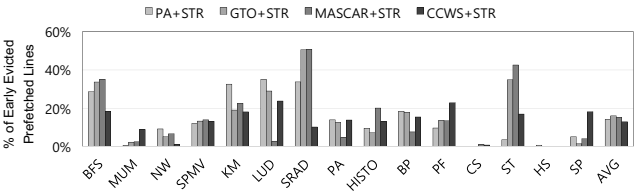


Figure 4. Early eviction ratio of STR prefetchers

We combined STR and SLD prefetchers with four of the existing warp scheduling techniques, Prefetch-Aware (PA) [10], GTO [14], MASCAR [15], and CCWS [14]. Figure 3 shows the speedup achieved with various combinations of scheduling and prefetching techniques. CCWS+STR performed best and shows 17.5% performance improvement compared to the baseline. PA+SLD shows the largest improvement of 9.4%.

We observe that SLD shows less speedup than STR in all schedulers except PA. Macro block prefetching in SLD is often too aggressive because the prediction is correct only when the access stride is in the range of two cache blocks which are grouped in the same macro block. Assuming 128 byte line size of our baseline GPU, SLD can cover strides less than 256 bytes. Meanwhile, the per-load stride detection used in STR can correctly predict an arbitrarily large stride value, and as shown in Table I the detected stride value is often larger than two cache lines. As a result STR performs better than SLD.

The small cache in GPUs makes prefetching more challenging. Under severe cache contention, prefetched cache lines might be evicted by other accesses. We measure the frequency of such *early evictions*. Figure 4 shows the fraction of early evictions over the total prefetches issued using the combinations of STR prefetcher and the warp schedulers. Here we counted only correctly predicted cache lines as part of the total prefetches issued. In CCWS+STR which performed the best, 13.0% of correctly prefetched cache lines are evicted earlier than demand request. In case of PA+STR, GTO+STR, and MASCAR+STR, the early eviction ratio of 14.2%, 16.0%, and 15.2% are obtained. This implies additional performance improvement can be

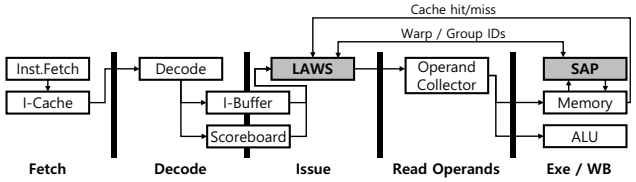


Figure 5. GPU architecture with APRES

possibly achieved by reducing early evictions. Thus the goal of this work is to reduce the early eviction rate while taking advantage of the cache access characterization of static loads.

IV. APRES Architecture

Our proposed technique, APRES, cooperatively operates warp scheduling with prefetching to improve cache efficiency. Figure 5 illustrates a GPU pipeline with two additional modules necessary to implement APRES.

The first module is the *Locality Aware Warp Scheduler (LAWS)* which we propose in this paper. LAWS is an advanced greedy scheduler which groups warps and then tries to orchestrate their progress such that a group of warps reach the same memory access instruction within a small time window. For this purpose, LAWS uses a scheduling queue which stores warp IDs in the priority order. The next issued warp is selected from the head of the queue. This greedy scheduling method implies that only a group of leading warps will be prioritized and executed ahead of the other remaining warps. Whenever any instruction is issued from the warp that is at the queue head, it may not be possible to continuously issue further instructions, particularly any read-after-write dependent instruction, from that warp due to pipeline delay. This latency depends on pipeline organization but without loss of generality we can assume 8 cycles [2]. Assuming 8 cycle latency, during the next 8 cycles, 7 more warps will be issued. As a result, 8 warps will be scheduled in a round robin fashion and all instructions are issued from these eight warps in a greedy manner. By prioritizing a group of eight warps, a smaller working set is accessed in a given time window and that will eventually reduce cache misses.

The first observation LAWS exploits is the fact that each static load across different warps exhibits the same cache behavior. For instance, if a load Y in warp A hits in cache, then the same load Y in warp B is also likely to hit in the cache if warp B is scheduled immediately after warp A. The problem is how to possibly group warps that will execute the same load instruction in the near future. LAWS solves this problem by grouping warps that executed the same static load *previously*. The key observation here is that GPU warps share the same kernel code, and tend to execute a similar sequence of instructions. In the above example, assuming warp A executed load X first and load Y next and another warp B executed load X, then we can expect warp B will execute load Y soon. We refer to the PC of load X as the LLPC (last load PC) of warp A and B. LAWS groups warp

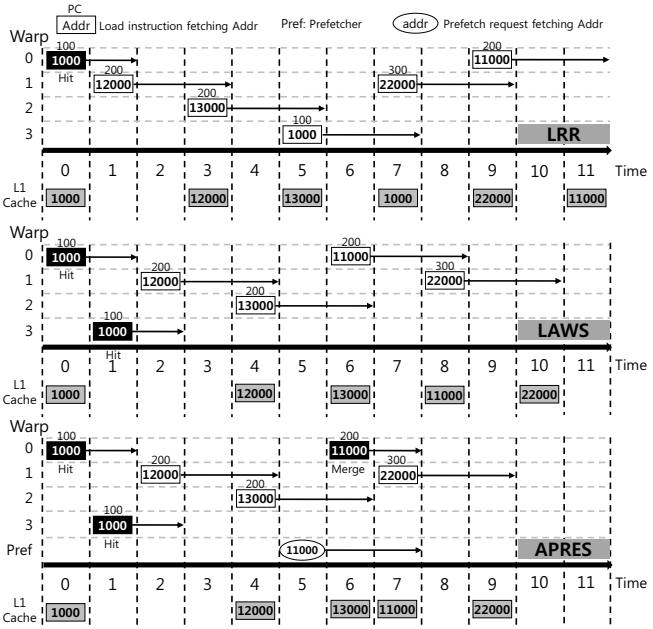


Figure 6. Warp execution with LRR, LAWS, and APRES

A and B when they share the same LLPC (load X in this example). As soon as the warp A issued load Y then warp B is grouped with warp A. Consequently, grouping with load X produces the same result as grouping with load Y.

The second observation, as discussed in Section III, LAWS exploits is that some loads have very high memory locality across different warps and such loads are likely to hit in cache. Since LAWS already groups warps based on the fact that they are likely to execute the same loads within a short time interval, it is then feasible to exploit the fact that if one load from a warp hits in the cache the same static load from the second warp will also hit in the cache.

The second part in Figure 6 shows an illustration how LAWS is able to group warps to reduce cache misses. For illustration purpose, we assume all addresses in the figure are mapped to the same cache block. Also, we assume all warps execute the same sequence of load instructions (load PC of 100, 200, and 300). When the address 1000 demanded by Warp 0 hits the cache (black rectangular), LAWS predicts that Warp 3 is going to execute the same PC 100 and it may also access the same data. Then, LAWS issues Warp 3 ahead of the other warps. We will describe shortly how LAWS is able to detect that Warp 0 and Warp 3 are about to execute the same static load instruction.

Our observations in Section III also showed that some loads do not exhibit any reuse. Hence, if a load from one warp exhibits little reuse, the same observation will hold for that load when executed in a second warp. So the last problem is how to determine the type of the next load (high locality or stride based references). To determine load type, we use the cache access result of the first warp that executes the load in a group (head warp) as a proxy of the cache

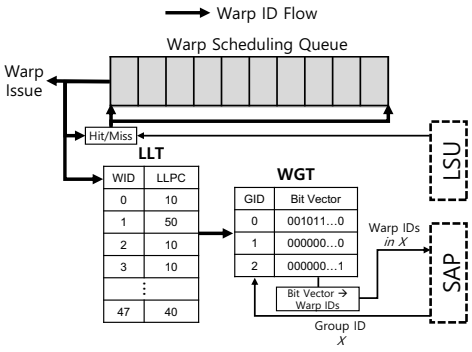


Figure 7. LAWS architecture

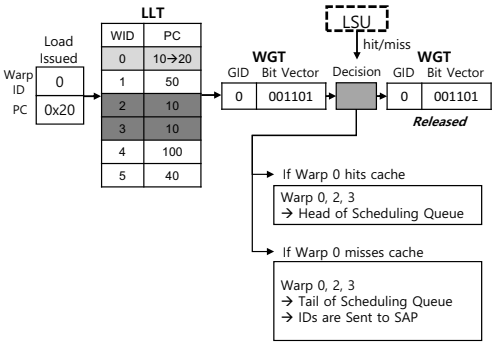


Figure 8. Example of warp grouping in LAWS

behavior of all warps in the group. This is possible because the load type is consistent across warps. If the head warp hits the cache, then LAWS moves all warps in the group to the front of the queue to execute them closely in time. For the high locality load the grouped warps should be prioritized because they expected to hit the cache. If the head warp misses the cache all warps in the group are moved to the tail of the queue to give them lowest priority.

Although the head warp missed the cache, there are opportunities to prefetch cache lines of the other warps in the same group when the grouped load is a strided type. Based on this observation, APRES proposes a grouped prefetching technique, namely *Scheduling Aware Prefetching (SAP)*. Once the head warp misses the L1 cache, SAP generates the prefetch requests for all the other warps in the same group. Utilizing stride behavior, SAP is able to predict the address accessed by other warps in the group, using the base address from the warp that missed in the cache. APRES targets to maximize the effect of prefetching by cooperation between SAP and LAWS. After SAP determines which warps are eligible to prefetch, LAWS gives prefetching target warps high priority. This scheduling policy minimizes early evictions by merging as many as possible demand requests to the corresponding MSHR entries generated by prefetching. Once a demand request is merged, the target warp can quickly access the prefetched cache line after the prefetched data arrives.

The bottom most part of Figure 6 illustrates how APRES combines LAWS and SAP together. In this example, Warp 1 and Warp 2 execute the load instruction at PC 200. They access the demand addresses of 12000 and 13000, respectively. When Warp 2 issues the load at PC 200, from the demand addresses of Warp 1 and Warp 2 a stride value of 1000 is computed. At the same time, LAWS groups Warp 0, 2, and 3 because they all have the same LLPC of 100. If the SAP recognizes that PC 200 is a striding load then it initiates a prefetch for Warp 0 and 3. In this example, the prefetcher generates a prefetch request to fetch the data in the memory address 11000 for Warp 0. After that, Warp 0, which is targeted for prefetch, is scheduled ahead of Warp 1. This execution makes the demand access of Warp 0 is merged to the prefetch request when it issues a load. If the warp scheduler issues Warp 1 ahead of Warp 0 as shown in the LRR case, the prefetched data may not be properly accessed by Warp 0 since it is evicted by the line fetched by Warp 1. APRES reduces cache eviction for the contended cache line, resulting in performance improvement.

A. Locality Aware Warp Scheduling

Figure 7 shows the structure and operation of the proposed LAWS. As discussed earlier, the warps which previously executed the same static long-latency load are required to be grouped together. For this purpose, the grouping logic tracks the last long-latency load instruction for each warp. We consider all global memory loads as long latency regardless they actually hit or missed the cache. This information is managed by the Last Load Table (LLT); LLT is indexed by the warp ID and each LLT entry records the last load PC (LLPC) of each warp. A warp group is formed when a long-latency load instruction is issued from a warp. When a long latency load is issued LAWS first reads LLPC of the issued warp from the LLT. To determine which warps should be grouped, LLT is searched to find all warps whose LLPC matches the LLPC of the issued warp.

The warps with a matching LLPC are grouped with the current warp and the grouping information is stored in Warp Group Table (WGT). Each WGT entry indicates a warp group and each entry is a 48-bit vector assuming 48 warps are accommodated per SM. Each bit that is set in the vector indicates that the corresponding warp ID is part of that group.

As soon as a long latency load is issued, all warps from the corresponding warp group are identified using the WGT. LAWS then schedules warps based on the feedback from Load Store Unit (LSU) and the SAP prefetcher. When a load is processed in the load store unit, warp ID of the current load, the associated warp group ID, and cache hit status of the load are sent to the scheduler. The scheduler first looks up WGT using the received group ID to find warps in the same group. If the access hits the cache, then LAWS moves all warps in the group to the head of the queue. If the access missed the cache, then LAWS moves all warps in the group to the tail of the queue. After group prioritization is done, the group information is invalidated from WGT.

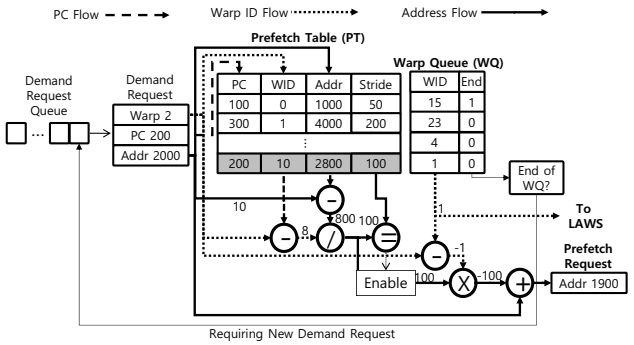


Figure 9. SAP architecture

Figure 8 illustrates an example of LLT management and warp grouping. When a load at 0x20 issued from Warp 0, LAWS searches LLT to find all warps whose LLPC is 0x10. In this illustration, Warp 2 and 3 are grouped together with Warp 0. Then the LLT entry of Warp 0 is updated to 0x20, which is the PC of the next issued load. Figure 8 shows two cache access scenarios after Warp 2 and 3 are issued. In this example Warp 0,2,3 are grouped according the last load PC. After Warp 0 hits the L1 cache Warp 2 and 3 are prioritized. Otherwise, Warp 0, 2, and 3 are moved to the tail of the scheduling queue.

LAWS also can trigger prefetch by sending relevant information to the SAP. If a cache miss is received from the load store unit, before clearing group information, the list of warps in the missed group is sent to the prefetcher. Prefetch requests are generated for a subset of warps in the group. LAWS then moves the received prefetch target warps to the queue head, so that these warps are prioritized.

B. Scheduling Aware Prefetching

Figure 9 describes the organization of the SAP prefetcher. SAP consists of a Prefetch Table (PT), a Warp Queue (WQ), and a demand request queue. PT stores necessary history of the previous load executions. The information includes the PC address, the warp ID which lastly issues the load, and the memory address. Also, each PT entry stores the stride value calculated from the two most recent memory addresses. WQ stores the warp IDs which are part of the group.

When a demand request misses in L1 cache, LAWS sends all of the warp IDs which belong to the same group of the warp which caused the miss to the prefetching engine. The warp IDs of the group are stored in WQ. The memory address of the missed request is stored in the demand request queue. If multiple requests are generated by a warp, then only the address requested by the lowest thread ID is stored in the demand request queue. SAP then initiates prefetch request generation.

Figure 9 shows an example after Warp 2 issues a load instruction (PC of 200) and generates a memory access request on memory address 2000. SAP searches a corresponding PT entry with the PC address. In this example, the warp ID in the entry found is 10 and the stride value is 100. First of all, SAP prefetches only when the inter-warp stride currently calculated matches to the value stored in the corresponding entry. The calculated stride in this example is 100; the previous address 2800 (from Warp ID 10) and the current address is 2000 (from Warp ID 2). Therefore, the current stride matches the stored stride value for warp 10 and thus prefetch requests are issued based on the stride value and base address stored in the table. If the stride values mismatch, then prefetching is not initiated at that instance and the stride in PT is replaced with the newly calculated value.

In the example of Figure 9, SAP generates a prefetch request for the warps in WQ. For the first warp (Warp 1), the difference of warp IDs is *1 - 2 = -1* and the stride is 100, and the prefetch address is calculated as *2000 + (-1) * 100 = 1900*. After SAP generates a prefetch request, it sends the prefetched warp ID back to LAWS so that LAWS can prioritize the warp.

Table II
HARDWARE COST OF APRES

Module	Cost
LAWS	4B×48 (LLT) + 48b×3 (WGT)
SAP	8B×32 (DRQ) + 1B×48 (WQ) + (4B+1B+8B+8B)×10 (PT)
Total	724 Bytes

C. Hardware Overhead

We estimate the additional hardware cost of APRES as shown in Table II. In the GPU architecture we use, it is assumed that a maximum of 48 warps are allocated per SM. To implement APRES, total 724 bytes for each SM are required. In LAWS, LLT requires 48 4-bytes elements to store the PC addresses. We set the number of entries in WGT to 3, which is equal to the number of pipeline stages from the issue stage to the execute stage. Three entries in WGT can cover all in-flight load instructions in the GPU pipeline.

SAP stores all memory addresses which are requested from a load instruction. Since a load can generate 32 requests at most when an uncoalesced access occurs, we set the size of the Demand Request Queue (DRQ) to 32. An entry in PT consists of a 4-byte element to store a PC address, an 1-byte element to store the warp ID, an 8-byte element to store a memory address, and an 8-byte element to store a stride. PT contains 10 entries to store the information about frequently executed load. WQ consists of 48 1-bytes elements, so that it requires 48 bytes.

We estimate the area overhead of APRES using CACTI [31], similar to the previous works [15], [32]. Compared to 32KB and 8-way set-associative L1 data cache, which is used in our evaluation, the additional hardware required in APRES is 2.06% of that cache. For address calculation, SAP requires 4 integer adders, 1 integer multiplier, and 1 integer divider. The overhead of those units is negligible compared to Fused Multiply-Add (FMA) units in CUDA cores.

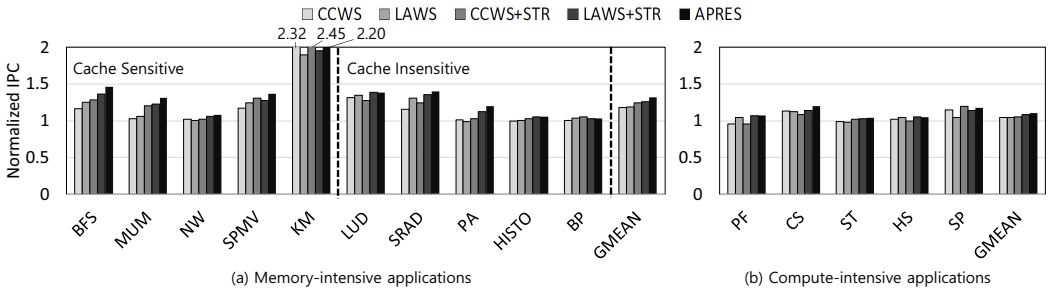


Figure 10. Performance comparison of LAWS and various prefetching techniques (Normalized to baseline)

Table III
SIMULATION CONFIGURATION

GPU Core	15 SMs, 1.4 GHz, SIMD width: 32 Max. concurrent 48 active warps
Warp Scheduler	LRR/GTO/CCWS/MASCAR/PA
Prefetcher	STR/SLD
L1 Data Cache	8-way, 32 KB, 128B line, 64 MSHRs
L2 Shared Cache	8-way, 768 KB, 128B line, 200 cycles latency
DRAM	6-partitioned, 924MHz, 440 cycles latency
Memory Request Merging	Request coalescing, Request merging in MSHR
Branch Control	Immediate post dominator
Baseline	LRR without prefetching
APRES	LAWS+SAP

Table IV
BENCHMARK APPLICATIONS

Application	Abbr.	Suites
Memory-intensive applications		
(a) Cache-sensitive applications		
Breath-First Search	BFS	Rodinia
MUMmerGPU	MUM	Rodinia
Needleman-Wunsch	NW	Rodinia
SParse-Matrix dense-Vector multiplication	SPMV	Parboil
KMeans	KM	Rodinia
(b) Cache-insensitive applications		
LU Decomposition	LUD	Rodinia
Speckle Reducing Anisotropic Diffusion	SRAD	Rodinia
PArticle filter	PA	Rodinia
HISTOgram	HISTO	Parboil
Back Propagation	BP	Rodinia
Compute-intensive applications		
PathFinder	PF	Rodinia
ConvolutionSeparable	CS	CUDA
Stencil	ST	Parboil
HotSpot	HS	Rodinia
ScalarProd	SP	CUDA

V. EXPERIMENTAL RESULTS

A. Methodology

We used GPGPU-sim v3.2.2 [33]. The detailed simulation configuration is described in Table III. Also, we used the power model from GPUWattch [34]. We select five cache-sensitive applications, five cache-insensitive applications, and five compute-intensive applications, respectively. The applications we use (listed in Table IV) are from CUDA Software Development Kit (SDK) [35], Parboil benchmark suite [36], and Rodinia benchmark suite [37].

B. Performance of APRES

Figure 10 compares IPC of the five techniques. For comparison, we used both CCWS and CCWS combined with the STR prefetching. CCWS improves IPC by 12.8% across 15 benchmarks. LAWS alone improves IPC by 14.0%. Combined with STR prefetching, as discussed in Section III-C, CCWS+STR shows 17.5% of IPC improvement. Even without prefetching, LAWS performed better than CCWS+STR in *LUD*, *SRAD*, *PF*, and *CS*. LAWS+STR, which is simply using the basic stride prefetching on top of LAWS, shows even higher improvement up to 18.8%. APRES which coordinates LAWS and SAP together, shows highest IPC improvement of 24.2%.

On cache-sensitive applications, APRES achieves 44.0% speedup while CCWS+STR provides 38.3% speedup. APRES outperforms CCWS+STR in all benchmarks, except KM. As discussed in Section III-B, KM suffers from excessive cache contention because of its huge working set size. CCWS reduces the number of active warps to increase effective cache size per thread. Without forcibly limiting the number of active warps, APRES solves this problem by more cleverly rearranging warps to be more cache friendly. But even with the improved scheduling it is still not as effective as completely eliminating the need for larger cache by reducing the number of active warps. Thus APRES achieves 119.6% of speedup, while CCWS shows 132.0% speedup, CCWS+STR shows 144.8% speedup.

On cache-insensitive applications, APRES achieves 20.4% performance improvement compared to the baseline GPU. Also, APRES shows 8.5% better performance than CCWS+STR for cache-insensitive applications. In the case of SRAD, LAWS gains 31.1% of speedup and APRES achieves 40.0% speedup. According to Table I, both high locality and strided loads coexist in SRAD. Thus, warp grouping and scheduling performed in LAWS is highly effective in improving L1 cache hits by separating two loads. More than 75.0% of requests have regular stride, as a result prefetching combined with LAWS achieves higher

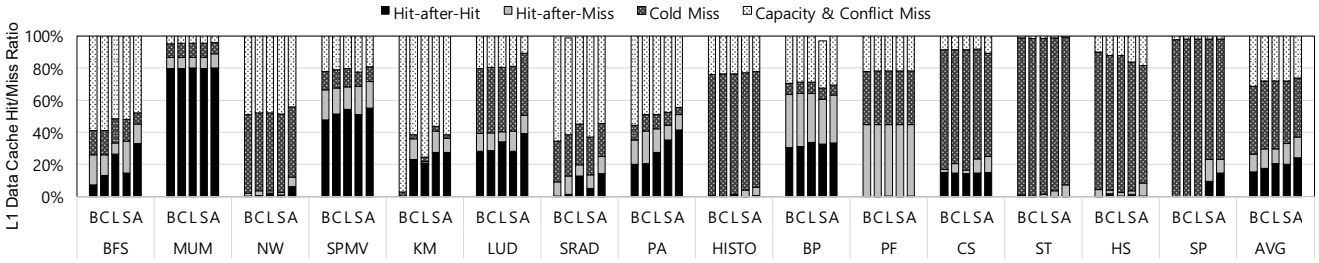


Figure 11. Cache hit and miss breakdown comparison (B: Baseline, C: CCWS, L: LAWS, S: CCWS+STR, A: APRES)

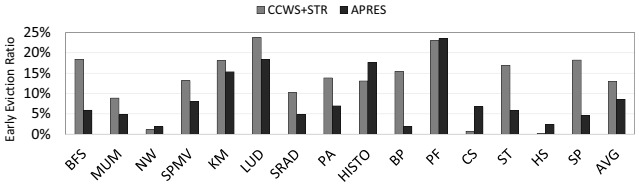


Figure 12. Early eviction ratio comparison

performance with APRES.

On compute-intensive applications, APRES shows more than 15% of performance improvement in CS and SP compared to the baseline GPU, while LAWS alone improves by 8.0%. Both CS and SP show regular memory access pattern and have low cache line reuse rate, thus prefetching of APRES contributed more to speedup than scheduling.

C. Impact on Cache

In this subsection, we analyze how APRES impact on the L1 data cache. We measured cache hit and miss ratios to observe how well the cache captures memory access locality. We divided cache hits into hit-after-hit and hit-after-miss. A hit-after-hit is a continuous cache hit. On the other hand, a hit-after-miss is a cache hit after a cache miss happens. Hit-after-hit ratio shows the effect of warp grouping by LLPC in LAWS. Also, we divided cache misses into cold miss and "capacity and conflict misses", using the same methodology presented in Section III.

Figure 11 shows the breakdown of L1 data cache utilization in the baseline GPU, CCWS only, LAWS only, CCWS+STR, and APRES. Across all 15 benchmarks, LAWS shows 3.0% higher hit-after-hit ratio than CCWS. In the applications which show frequent cache hits, such as BFS, SPMV, LUD, SRAD, and PA, LAWS shows more than 10% of hit-after-hit ratio compared to CCWS. Also, LAWS shows higher total cache hit ratio in these applications. The warp grouping method in LAWS successfully improves the cache hit ratio by increasing continuous cache hits. This effect is also shown in APRES. APRES shows the best hit-after-hit ratio, which is 4.1% better than CCWS+STR.

Moreover, for all benchmarks, APRES shows 10.3% less cache miss rate than the baseline GPU and 5.9% less cache miss rate than CCWS+STR. On cache-sensitive applications, APRES reduces average cache miss by 14.5% and 3.9% compared to the baseline and CCWS+STR, respectively.

APRES reduces capacity and conflict misses through LAWS and reduces cold miss through SAP. We can further analyze the additional cache miss reduction achieved by SAP comparing LAWS and APRES results. There is a clear trend that on top of miss reduction achieved with LAWS, cold misses further reduced with SAP. For instance, in the BFS application, APRES reduces the capacity and conflict misses by 11.0%. As discussed in Section III, BFS shows frequent inter-warp cache hits, and LAWS makes the necessary cache line alive for a long time by prioritizing warps that will access the cache line. SAP further reduced cold misses of BFS less than half of the baseline. APRES achieves miss reduction of 19.0% compared to the baseline, resulting in speedup of 46.0%.

In cache-insensitive application category, APRES is effective in reducing capacity and conflict misses of LUD, SRAD, and PA. As a result, large speedup up to 40.1% (of SRAD) is achieved. In compute-intensive applications, APRES reduces the cache misses in CS and SP. We clearly see the speedup of CS and SP is due to reduced cold misses by the effect of prefetching.

D. Impact on Early Eviction

We measured early eviction ratio with CCWS+STR and APRES as shown in Figure 12. As presented in Section III-C, early eviction ratio is defined as the fraction of cache lines evicted before read by demand access. The average early eviction ratio of APRES is 8.6% and CCWS+STR is 13.0%. APRES reduced early eviction by 4.4%. In a vast majority of benchmarks, APRES consistently shows lower early eviction ratio than CCWS+STR. On cache-sensitive benchmarks, APRES outperforms CCWS+STR except NW. In case of BFS, early eviction ratio is reduced from 18.4% with CCWS+STR to 5.9% with APRES, indicating prefetch efficiency is drastically improved. On cache-insensitive applications, APRES shows 9.9% of early eviction ratio, which is 5.4% less than CCWS+STR. In the compute-intensive application category, APRES is also effective in reducing early evictions, particularly for the ST and SP benchmarks, resulting in up to 17.2% (SP) speedup is observed.

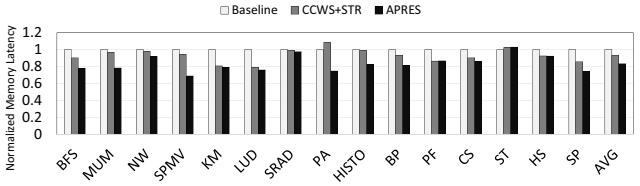


Figure 13. Average memory latency comparison (Normalized to baseline)

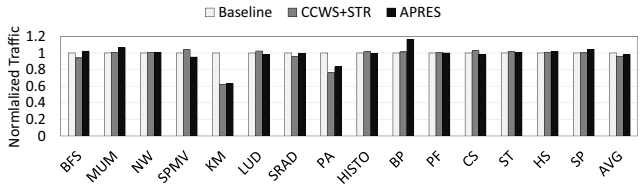


Figure 14. Data traffic comparison (Normalized to baseline)

E. Impact on Memory Latency and Traffics

Figure 13 shows the ratio of average memory latency of the baseline, CCWS+STR, and APRES. APRES reduces the memory latency by 16.5% compared to the baseline, and by 9.7% compared to CCWS+STR. In the applications which APRES achieves the significant speedup, APRES significantly reduces the memory latency. As APRES results in more cache hits, memory system is less congested leading to latency reduction. In case of BFS, APRES shows 22.0% and 9.6% less memory latency than the baseline and CCWS+STR, respectively. In KM, both APRES and CCWS+STR reduce the memory latency by 19.2% and 20.8% compared to the baseline. On SPMV, APRES reduces the memory latency by 31.8%, which is due to increased cache hits as shown in Figure 10. Considering cache miss results as shown in Figure 11, we find that reducing capacity and conflict misses by LAWS also reduces memory access latency. In case of PA, CS, and SP, the memory latency of APRES is reduced because of SAP not because of LAWS.

As GPU throughput is often limited by memory bandwidth, ineffective prefetches may lead to performance degradation. To analyze the impact of prefetching on data traffics, we measured the amount of data moved from memory to SM measured as the number of data bytes transferred. Figure 14 shows relative data traffics with the baseline, CCWS+STR, and APRES. It appears the average traffics is decreased by 3.8% with CCWS+STR, and by 2.1% with APRES. Both the STR prefetcher and SAP in APRES adopt adaptive scheme that issues prefetch requests only when the detected stride value shows regular pattern and the address prediction is likely to be correct. As a result, ineffective prefetches are minimized. As shown in the figure, the amount of traffics appears to be similar to the baseline on both CCWS+STR and APRES. In BP, APRES increases traffics by 16.4%, but as shown in Figure 10, negative performance impact is not observed.

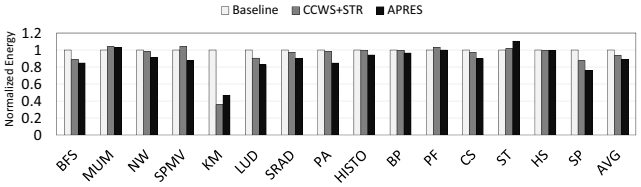


Figure 15. Dynamic energy consumption normalized to Baseline

F. Impact on Energy Consumption

As moving data is an energy hungry operation in GPU, improving cache efficiency also improves energy efficiency. We simulated energy impact of APRES. Additional energy consumption of new blocks for APRES is also modeled. In fact, the overhead of additional blocks was less than 3% of the total energy. As presented in Figure 15, APRES reduces 10.8% of the dynamic energy compared to the baseline. APRES reduces the energy consumption more than 15% in BFS, KM, and SP. In case of ST, APRES increased energy consumption because many prefetches are not effective and as a result memory traffics and power consumption are increased. However, even in this worst case the increased energy consumption was less than 10%, and simply turning off the prefetcher will prevent any energy waste due to prefetching.

VI. Related Work

Warp Scheduling Techniques: Warps in an SM are scheduled by warp schedulers and the performance is influenced by the scheduling policy. Loose Round Robin (LRR) gives the equal priority to all warps [33]. While LRR is good in terms of fairness, this scheduling policy tends to make all warps reach the long latency memory operations roughly at the same time. Thus, it may cause the performance degradation due to memory contention and hardware underutilization [17]. To overcome the drawbacks of LRR scheduling, the two-level warp scheduler was introduced [13]. The main contribution of two-level scheduling is that warps are divided into multiple groups. A warp group is swapped with another group of warps when all warps in the currently active group reaches the long latency operations. Thus, two-level scheduling can hide the long stalls since another groups are issued and kept operated until warp stall of a warp group is resolved. There are many schedulers derived from two-level scheduling [8], [13], [17]

Recent studies have proposed various warp scheduling techniques to improve the efficiency of memory system utilization. Rogers et al. proposed a Cache-Conscious Wavefront Scheduling (CCWS) to leverage the intra-warp locality for efficient cache utilization [14]. We evaluated CCWS in Section III-C and Section V. They also designed another warp scheduling method which focuses to predict the cache utilization in memory divergence situation [16]. Jog et al. introduced another scheduling scheme improving the performance by reducing the cache contention and enhancing

the DRAM bank-level parallelism [9]. Sethia et al. proposed a warp scheduling and a cache re-execution method for the efficient utilization of memory hardware resource to improve the memory-intensive workloads [15]. Lee et al. recently introduced two warp scheduling techniques to improve the performance balance between faster and slower warps, which are critical to the overall performance [11], [12].

Prefetching: Hardware prefetching schemes have been widely adopted in CPUs for reducing effective memory latency. There exist the stride prefetching and the stream prefetching schemes. In case of stride prefetching, it detects the regular memory access pattern and predicts the future memory strides to be accessed [25], [26], [27]. It stores the prefetched lines to the cache memory. On the other hand, the stream prefetching schemes utilizes the external buffers to store the prefetched data [28], [29], [30]. Different from those two schemes, a probabilistic model of prefetching was also introduced [38].

The prefetching studies for GPGPUs were introduced by rising of GPUs's utilization for general purpose applications. Lee et al. proposed a many-thread aware prefetching scheme considering massively parallel processing on GPUs [20]. The scheme consists of the inter-thread prefetching and per-warp stride prefetching. Also, Sethia et al. introduced a new approach of per-warp stride prefetching to reduce the power consumption of GPUs. It helps to save the energy by reducing the number of active warps [21]. To address the poor performance of irregular memory access, Lakshminarayana et al. designed a prefetching scheme which is especially focused on the graph applications [39]. The methodology utilized the spare register file to store the prefetched data. Moreover, it focused to exploit the invalid thread operations by changing the valid load instruction executions.

Combined Warp Scheduling and Prefetching: Recently, the approaches about combined warp scheduling and prefetching scheme were proposed to improve the performance of GPUs [9], [10]. It boosts up the GPU performance by scheduling the warp groups, especially considering the spatial locality to reduce the stalls from memory pipeline. Also, the schemes choose the warp groups, which access the different DRAM banks to avoid the memory traffic bias problem. The approaches achieve the performance improvement based on such efficient access of memory subsystem. Also, their prefetching scheme brings the data in the same row in the DRAM bank. Combining those techniques, they succeed to operate the prefetching with little overhead of memory traffic.

However, the prefetching techniques which are introduced in those studies did not show the significant performance improvement. They did not address early evictions of prefetched data, while APRES target to reduce early eviction by coordinating LAWS and SAP. Also, it is found that more than 60% of late prefetching still exists in their experimental results. We focus on those problems, and propose a solution with more harmonic execution of the warp schedulers and the hardware prefetchers considering the warp scheduling and cache access status. The proposed architecture gives the high scheduling priorities to the warps which are expected to make cache hits or those which are selected as the prefetching targets. The proposed prefetcher supports this functionality by sending the IDs of warps after it generates the prefetch requests for them.

GPU Cache Utilization: Since the cache size of GPU is not sufficient for massive number of threads, many researches proposed the techniques to utilize the cache more efficiently [19], [40], [41], [42]. First, several researches proposed the cache bypassing techniques. Chen et al. proposed an adaptive cache management techniques [19]. They proposed a new technique which combines the active warp throttling and cache bypassing. Their cache bypassing technique is activated if the memory traffic congestion is detected in the Network-On-Chip (NOC) level. The adaptive cache management technique improves the GPU performance by throttling the number of active warps to fit the working set size to the data cache and properly skip the cache utilization. This mechanism reduces the cache contentions. Jia et al. proposed another cache utilization technique, called MRPB [40]. This technique contains a novel memory request ordering and a cache bypassing. MRPB finds the conflicts between memory requests and prevents them by reordering memory requests. Some of reordered memory requests bypass the data cache if the stall is predicted. Tian et al. designed a novel cache bypassing for GPUs [41]. The proposed technique determines the cache bypassing for each static load. Xu at al. proposed a scheduling technique to minimize memory and cache contention when multiple applications are co-located in the same SM [42].

In addition, several recent studies have proposed various novel on-chip local memory structures fit to the GPU application characteristics. Gebhart et al. proposed a new on-chip local memory structure [43]. It has a form of unified memory structure between data cache, scratchpad memory, and register file. The portion of each structure is determined dynamically when a new GPU kernel is launched to GPU. Such a portion is estimated based on the number of active threads, dynamic instruction count, and the amount of off-chip device memory accesses. Wang et al. introduced a new cache management technique, called DaCache [44]. It consists of a new warp scheduling technique, a cache replacement policy, and a cache bypassing method. Dacache detects the loads which cause the memory divergence and prioritizes the warps which issue the coherent load over the warps which issue divergent loads. Komuravelli et al. proposed a new on-chip local memory structure, Stash [45]. Stash employs the advantages of both data cache and scratchpad memory. Stash is globally addressable like data cache and also available to reuse the data implicitly. With this flexibility, it improves the GPU performance.

Without adding or redesigning GPU memory hierarchy, APRES aims to fully utilize the existing data cache by combining the warp scheduling and data prefetching by better exploiting to the characteristics of static loads on GPU applications.

VII. Conclusion

This paper presents two important observations regarding the cache access behavior of GPU loads. Using a detailed per-load characterization we show that some loads access only a small range of memory space, thus exhibit strong locality. And the second category of loads have a large memory footprint that is frequently accessed using a stride access pattern. The characteristics are unique to each static load regardless of warp IDs. We exploit these observations to improve memory access bottlenecks in GPUs. We propose the APRES architecture which consists of a new locality aware warp scheduling and a new inter-warp prefetching technique. The warp scheduler tries to group warps such that all the warps in the group execute a particular load instruction within a short window of time, if that load instruction is deemed to exhibit strong locality. Thus the proposed warp scheduler enables multiple consecutive accesses to a cache line before it is evicted. For loads that have a large memory footprint the prefetcher tracks the inter-warp striding behavior and is able to issue prefetches for different warps which will execute the same static load which is currently being issued. These two modules adaptively cooperate by sharing information about prefetching target warps. APRES achieves significant performance improvement over any combinations of existing warp scheduling and prefetching techniques.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2015R1A2A2A01008281 and No. NRF-2015R1C1A1A01053844), by SIC R&D Center, LG Electronics Inc., and by the following grants: DARPA-PERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211, NSF-0834798. W. W. Ro is the corresponding author.

References

[1] N. Chatterjee, M. O'Connor, G. Loh, N. Jayasena, and R. Balasubramonia, "Managing dram latency divergence in irregular gpgpu applications," in *High Performance Computing, Networking, Storage and Analysis, International Conference for (SC14)*, Nov 2014, pp. 128–139.

[2] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, 2009, pp. 152–163.

[3] O. Kayiran, A. Jog, M. Kandemir, and C. Das, "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)*, Sept 2013, pp. 157–166.

[4] S.-Y. Lee and C.-J. Wu, "Characterizing the latency hiding ability of gpus," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '14)*, 2014, pp. 145–146.

[5] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A gpu pre-execution approach for improving latency hiding," in *2016 IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA '16)*, March 2016, pp. 163–175.

[6] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*, 2016.

[7] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in gpus," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. New York, NY, USA: ACM, 2012, pp. 15–24.

[8] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*, 2011.

[9] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. New York, NY, USA: ACM, 2013, pp. 395–406.

[10] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. New York, NY, USA: ACM, 2013, pp. 332–343.

[11] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. New York, NY, USA: ACM, 2015, pp. 515–527.

[12] S.-Y. Lee and C.-J. Wu, "Caws: Criticality-aware warp scheduling for gpgpu workloads," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*, 2014, pp. 175–186.

[13] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. New York, NY, USA: ACM, 2011, pp. 308–317.

[14] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 72–83.

[15] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*, Feb 2015, pp. 174–185.

[16] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. New York, NY, USA: ACM, 2013, pp. 99–110.

[17] M. K. Yoon, Y. Oh, S. Lee, S. H. Kim, D. Kim, and W. W. Ro, "Draw: investigating benefits of adaptive fetch group size on

gpu," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '15)*, March 2015, pp. 183–192.

[18] NVIDIA, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.

[19] X. Chen, L.-W. Chang, C. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, Dec 2014, pp. 343–355.

[20] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*, Dec 2010, pp. 213–224.

[21] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 73–82.

[22] NVIDIA, *NVIDIA GeForce GTX 980: Featuring Maxwell, The Most Advanced GPU Ever Made*, 2014.

[23] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. New York, NY, USA: ACM, 2013, pp. 86–98.

[24] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the tradeoffs between software-managed vs. hardware-managed caches in gpus," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '14)*, March 2014, pp. 231–242.

[25] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, no. 5, pp. 609–623, May 1995.

[26] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO '92)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 102–110.

[27] Y. Liu and D. Kaeli, "Branch-directed and stride-based data cache prefetching," in *Computer Design: VLSI in Computers and Processors, Proceedings., 1996 IEEE International Conference on (ICCD '96)*, Oct 1996, pp. 225–230.

[28] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*. New York, NY, USA: ACM, 1990, pp. 364–373.

[29] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 24–33.

[30] S. Sair, T. Sherwood, and B. Calder, "A decoupled predictor-directed stream prefetching architecture," *Computers, IEEE Transactions on*, vol. 52, no. 3, pp. 260–276, 2003.

[31] S. Thoziyoor, N. Muralimanohar, and J. H. Ahn, "Cacti 5.1," Hewlett-Packard Laboratories, Tech. Rep., 2008.

[32] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient gpus through register compression," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. New York, NY, USA: ACM, 2015, pp. 502–514.

[33] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*, April 2009, pp. 163–174.

[34] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. New York, NY, USA: ACM, 2013, pp. 487–498.

[35] NVIDIA, *NVIDIA CUDA SDK 4.0*.

[36] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009. IEEE International Symposium on Workload Characterization (IISWC 2009)*, Oct 2009, pp. 44–54.

[38] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*. New York, NY, USA: ACM, 1997, pp. 252–263.

[39] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*, Feb 2014, pp. 614–625.

[40] W. Jia, K. A. Shaw, and M. Martonosi, "Mrpb: Memory request prioritization for massively parallel processors," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*, Feb 2014, pp. 272–283.

[41] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive gpu cache bypassing," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU 2015)*. New York, NY, USA: ACM, 2015, pp. 25–35.

[42] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*, 2016.

[43] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 96–106.

[44] B. Wang, W. Yu, X.-H. Sun, and X. Wang, "Dacache: Memory divergence-aware gpu cache management," in *Proceedings of the 29th ACM International Conference on Supercomputing (ICS '15)*. New York, NY, USA: ACM, 2015, pp. 89–98.

[45] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. New York, NY, USA: ACM, 2015, pp. 707–719.