CTA-aware Prefetching for GPGPU

Hyeran Jeon, Gunjae Koo, Murali Annavaram

Computer Engineering Technical Report Number CENG-2014-08

Ming Hsieh Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562

October 2014

CTA-Aware Prefetching for GPGPU

Hyeran Jeon Gunjae Koo Murali Annavaram
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA
{hyeranje, gunjae.koo, annavara}@usc.edu

Abstract

In this paper, we propose a thread group-aware stride prefetching scheme for GPUs. GPU kernels group threads into cooperative thread arrays (CTAs). Each thread typically uses its thread index and its associated CTA index to identify the data that it operates on. The starting base address accessed by the first warp in a CTA is difficult to predict, since that starting address is a complex function of thread index and CTA index and also depends on how the programmer distributes input data across CTAs. But threads within each CTA exhibit stride accesses. Hence, if the base address of each CTA can be computed early, it is possible to accurately predict prefetch addresses for threads within a CTA. To compute the base address of each CTA, a leading warp is used from each CTA. The leading warp is executed early by pairing it with warps from currently executing leading CTA. The warps in the leading CTA are used to compute the stride value. The stride value is then combined with base addresses computed from the leading warp of each CTA to prefetch the data for all the trailing warps in the trailing CTAs. Through simple enhancements to the existing two-level scheduler, prefetches can be issued sufficiently ahead of time before the demand requests. CTA-aware prefetch predicts addresses with over 99.27% accuracy and is able to improve GPU performance by 10%.

1. Introduction

Long latency memory operation is one of the most critical performance hurdle in any computation. Graphics processing units (GPUs) rely on dozens of concurrent warps for hiding the performance overhead of long latency operation by quickly context switching among all the available warps. When warps issue a long latency load instruction they are descheduled to allow other ready warps to issue. Several warp scheduling methods have been proposed to efficiently select ready warps and to deschedule long latency warps so as to minimize wasted cycles of a long latency operation. For instance, recently proposed two-level schedulers [13] employ two warp queues: pending queue and ready queue. Only warps in the ready queue are considered for scheduling and when a warp in the ready queue encounters a long latency operation, such as a load instruction, it is pushed out into the pending queue. Any ready warp waiting in the pending queue is then moved to the ready queue.

In spite of these advancements, memory access latency is still a prominent bottleneck in GPUs, as has been identified in prior works [7, 19, 28, 33]. To tackle this challenge researchers began to adopt memory prefetching techniques, which have been studied extensively in the CPU domain, to the GPU domain. Recent GPU-centric prefetching schemes [16, 17, 20, 21, 29] can be categorized into three categories: intra-warp stride prefetching, inter-warp stride prefetching, and next line prefetching.

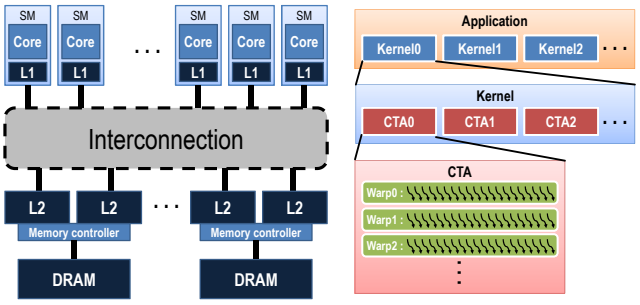
1.1 GPU Hardware and Software Execution Model

Before describing the applicability and limitations of these three approaches and the need for the proposed prefetching scheme, we first provide a brief overview of GPU architecture and application execution model. Figure 1a shows the GPGPU hardware architecture composed of multiple streaming multiprocessors (SMs) and memory partitions [27]. Each SM has 32 single instruction multiple thread (SIMT) lanes where each SIMT lane has its own execution units, also called a CUDA core. Each SM is associated with its own private memory subsystem, a register file and level-1 texture, constant, data and instruction caches. SMs are connected to level-2 cache partitions that are shareable across all SMs via an interconnection network. L2 cache banks connect to a larger DRAM-based global memory through one or more memory controllers where each controller is associated with one or more L2 cache partitions. If requested data is serviced by an external DRAM chip, the latency, which varies with memory traffic congestion, is hundreds or even thousands of GPU core cycles [37].

The GPU software execution model is shown in Figure 1b. A GPU application is composed of many kernels, which are basic task modules that exhibit significant amount of parallelism. Each kernel is split into groups of threads called thread blocks or concurrent thread arrays (CTA). A CTA is a basic workload unit assigned to an SM in a GPU. Threads in a CTA are sub-grouped into a warp, the smallest execution unit sharing the same program counter. In our baseline hardware, a warp contains 32 threads. For memory operations, a memory request can be generated by each thread and up to 32 requests are merged when these requests can be encapsulated into a cache line request. Therefore, only one or two memory requests can be generated if requests in a warp are highly coalesced.

1.2 CTA distribution

GPU compilers estimate the maximum number of concurrent CTAs that can be assigned to an SM by determining the resource usage information of each CTA, such as the register file size and shared memory usage – the available resources within an SM must meet or exceed the cumulative resource demands of all the CTAs assigned to that SM. Furthermore, the GPU hardware itself places a limitation on the number of warps that can be assigned to each SM. For example, NVIDIA Fermi can run up to 48 warps in an SM. Thus if a kernel assigns 24 warps per CTA, each SM can accommodate up to two concurrent CTAs. For load balancing, current GPUs assign a CTA to each SM in a round-robin fashion until all SMs are assigned up to the maximum concurrent CTAs that can be accommodated in an SM. Once each SM is assigned the maximum allowed CTAs then future allocation of CTAs to an SM are purely demand-driven. A new CTA is assigned to an SM only when an existing CTA on that SM finishes execution.



(a) GPGPU architecture (b) GPU application structure

Figure 1: GPGPU hardware and software architecture

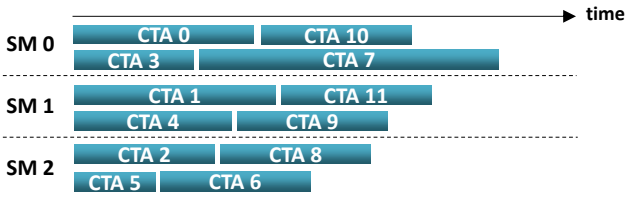


Figure 2: Example CTA distribution across SMs where each SM runs two concurrent CTAs and the kernel has 12 CTAs. Each rectangle presents the execution of the denoted CTA.

Figure 2 shows an example CTA distribution across three SMs. Assume that a kernel consists of 12 CTAs each SM can run two concurrent CTAs. In the beginning of the kernel execution, SM 0, 1, and 2 are assigned two CTAs, one at a time in a round-robin fashion; CTA 0 to SM 0, CTA 1 to SM 1, CTA 2 to SM 2, CTA 3 to SM 0 and so on. Once the six CTAs are first allocated to all the SMs, the remaining six CTAs are assigned whenever any of the assigned CTAs terminates. When CTA 5 execution is finished first, CTA 6 is assigned to SM 2. The next CTA to finish execution is CTA3 and thus CTA 7 is assigned to SM 0. Therefore, CTA assignments to an SM are determined dynamically based on CTA termination order.

2. Limitations of Prefetches in GPUs

With the above background, we now describe the limitations of existing GPU prefetchers and how one can overcome these challenges.

2.1 Intra-warp stride prefetching

Prefetching of strided data requests is a basic prefetching method that was explored for CPUs and has been shown to be effective when array data is accessed with regular indices in a loop [5, 12]. In the context of a GPU application if each thread within a warp loads array data from memory repeatedly in a loop then stride prefetching is initiated to prefetch data for future loop iterations of each thread. Since each prefetch targets the load instruction of a future loop iteration of the same thread within the warp, this approach is called *intra-warp stride prefetching*. Intra-warp stride prefetching was recently proposed for graph algorithms running on GPUs [20]

The effectiveness of intra-warp stride prefetching depends on the presence of load instructions that are repeatedly executed in loops to access array structures. But there is a growing trend towards replacing deep loop operations in a GPU applications with parallel thread operations with just a few loop iterations in each thread. Thus deep loop operations are being replaced with thread-level parallelism with reduced emphasis on loops.

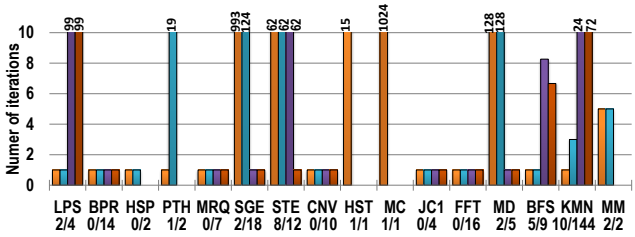


Figure 3: Average number of iterations for load instructions in a kernel. Repeated load instructions / total load instructions (by PC) under names of benchmarks

Figure 3 shows the average number of times the four common load instructions executed in a given warp in each of the selected benchmarks (benchmarks and simulation methodology are described in detail in Section 5). We counted how often each load instruction, distinguished by the PC value, was executed in a warp. If a load instruction is part of a loop body then that PC would have repeatedly appeared in the execution window. Also the total number of load instructions that appeared as part of a loop body over the total load instructions is also shown for each benchmark under the benchmark name on X-axis. The results show that only a few loads appear in a loop body.

These results show that when a loop intensive C program is ported to CUDA (or OpenCL), loops are reduced to leverage massive thread level parallelism. For example, whereas matrix multiplication A×B requires rowA×colA×colB iterations for each load with a naive serial program, each load instruction is executed 5 times in a kernel for the GPU program. This observation has also been made in a prior study that showed that deep loop operations are seldom found in GPU applications [24, 25].

CUDA and OpenCL favor vector implementation over loops because of its scalability. By enabling thread level parallelism the software becomes more scalable as the hardware thread count increases. For instance, if the number of hardware threads double then each hardware thread is assigned half the number of vector operations without re-writing the code. Favoring thread parallelism, over loops, results in loss of opportunities for intra-warp stride prefetching. Thus a prefetch scheme should not only capture iterative loads appearing in a load, but it should also target loads that are not part of any loop body.

2.2 Inter-warp stride prefetching

The stride detector based on the same PC per warp can be extended to an inter-warp stride prefetcher [21, 29] to account for more thread level parallelism. If regular offsets of memory addresses are detected between different warps, then inter-warp prefetching detects base address and stride value across different warps based on warp-id. Thus inter-warp stride prefetcher issues prefetches for a future warp from a current warp using the base address and warp-id differences.

The CTA distribution algorithms employed in current GPUs limits applicability of inter-warp stride prefetching to warps within a CTA. As shown in Figure 2, SMs are not assigned consecutive CTAs. Thus within a CTA all the warps are able to see stride accesses but the prefetcher is unable to prefetch across CTAs assigned to the same SM. Thus accurate prefetch is limited to warps within a CTA. Inter-warp prefetching will be more effective if there are a larger number of warps in a CTA. Table 1 shows CTA statistics for the 15 benchmarks used in our study. The column titled warps/CTA

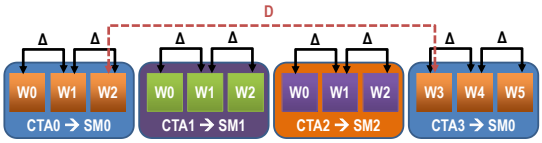


Figure 4: An example of distance of memory addresses among warps (W#). Each CTA contains 3 warps and strides of memory addresses of warps within a CTA are the same. However, distance between W2 of CTA0 and W3 of CTA3 is different even though they are consecutive warps in an SM.

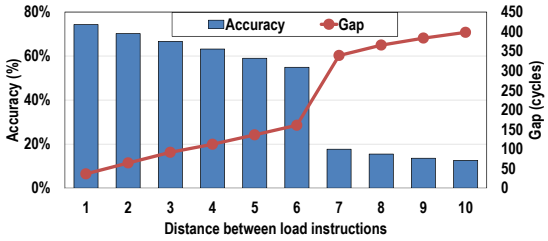


Figure 5: Accuracy with stride-based inter-warp prefetch and cycle gaps by distance of load instructions

shows the average number of warps in each CTA. Most benchmarks have fewer than 10 warps per CTA and at most 16 warps are assigned in a CTA, limiting the effectiveness of inter-warp prefetching. Having more CTAs than having warps per CTA also improves scheduling ability since each CTA is assigned to execute on one SM.

Application	Tot. CTAs	Conc. CTAs	Warps /CTA	Application	Tot. CTAs	Conc. CTAs	Warps /CTA
LPS	100	8	4	HST	4370	8	2
BPR	4096	6	8	MC	128	8	4
HSP	1849	3	8	JC1	16	6	8
PTH	463	6	8	FFT	256	8	2
MRQ	512	3	16	MD	48	4	8
SGE	528	8	4	BFS	1954	3	16
STE	1024	8	4	KMN	512	8	16
CNV	18432	8	2	MM	50	4	8

Table 1: CTA statistics

Figure 4 shows an example of disruptions to regular striding between two consecutive warps when these warps straddle CTA boundaries. In this example, each CTA has three warps and SM 0 is assigned CTA 0 and CTA 3. The stride of memory addresses between consecutive warps within the CTA is Δ. However, the distance of memory address between warp 2 and warp 3 is not Δ because they straddle CTA boundaries. With a simple inter-warp stride prefetching wrong prefetches are issued for warp 3. Note that a well designed stride prefetcher detects this change and eventually corrects the new base address and stride for CTA 3. But this correction disturbs the prefetch timeliness which is critical for performance. Thus every time a correction is made to the stride prefetcher the timeliness of prefetching is compromised.

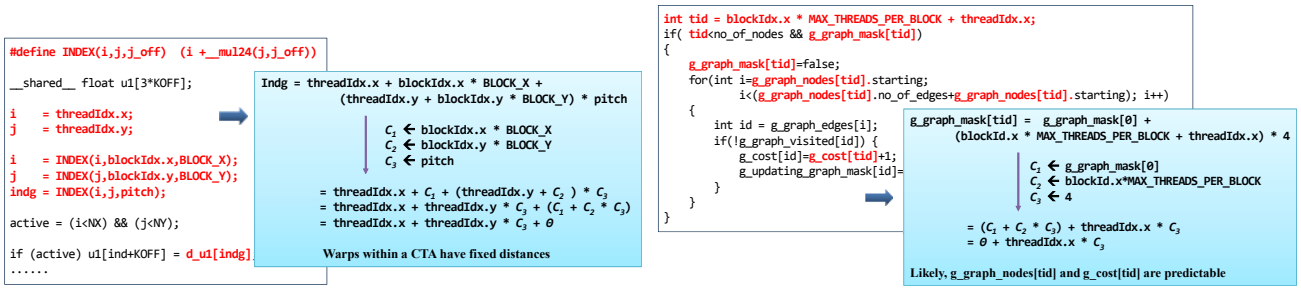
Figure 5, shows the tradeoff between prefetch timeliness and prefetch accuracy for the simplest and most stride-friendly benchmark matrixMul. From Table 1 we can infer that matrixMul (labelled as MM in the table) has 50 CTAs and each CTA has 8 warps. The X-axis in the figure shows how far ahead a prefetch is issued. A value of one means that warp W_n prefetches for W_{n+1}, a distance of two means that warp W_n prefetches for W_{n+2}. When the distances are short then accuracy is high. Accuracy is defined as the fraction of prefetch addresses that match the demand addresses that are generated later. The primary Y-axis shows the accuracy. The line plot shows the prefetch gap, which is the number of cycles between between prefetch and demand fetch, on the secondary Y-axis. If the distance is just one, W_n prefetches for W_{n+1}, then the number of cycles between the prefetch and demand fetch is just few tens of cycles. Since global memory access takes hundreds of cycles a short prefetch gap cannot hide the access latency. In order to increase the gap between prefetch and demand fetch one has to increase the distance. But as we move along the X-axis the prefetch accuracy drops gradually and then suffers a steep drop at a distance of seven. As the distance increases one has to cross CTA boundaries between the prefetch and demand fetch. Since MM has 8 warps per CTA, at the distance of seven every prefetch crosses the CTA boundary; note that we need two warps to be executed first to compute the base address and stride before issuing a prefetch. Once the prefetch crosses the CTA boundary the accuracy drops dramatically.

2.3 Next-line prefetching

The last category of GPU prefetching is next line prefetching, which fetches the next one or two consecutive cache lines alongside the demand line on a cache miss. The basic next line prefetch is agnostic to application access patterns and hence it leads to a significant increase in wasted bandwidth. Next line prefetching in conjunction with warp scheduling polices for GPUs was proposed in [16, 17]. The proposed warp scheduler assigns consecutive warps to different scheduling groups. The warp in one scheduling group can prefetch data for the logically consecutive warp which will be scheduled later in different scheduling group. While the cache miss rate is in fact reduced with this scheme, as we show later in our results section, prefetches are issued too close to the demand fetch, resulting in small performance improvements.

3. Where Did My Strides Go?

In this section we provide some insights into how GPU execution model perturbs stride access patterns that may be seen at the application level. Figure 6 shows two example codes, from the LPS and BFS benchmarks, that do not use loop iterations to execute loads. The bold code lines (also shown in red color) of the left hand side code box are the CUDA code statements that calculate the indices used in accessing the array data (array d_u1 in LPS, and arrays g_graph_mask, g_graph_nodes, g_cost in BFS). The right hand side shaded blue box represents the corresponding equation to show how the array index will be eventually computed using various parameters. Many GPU kernels use thread id and block id (also called CTA id) to compute the index values for accessing the data that will be manipulated by each thread. CTA parameters such as BLOCK_X and BLOCK_Y are compile-time known values that can be treated as fixed values across all the CTAs within each kernel. Parameters such as blockId.X and blockId.Y are CTA-specific values that are constant only across all the threads within a CTA. Thus load address computations rely on a mix of parameters, that are constant across all CTAs, such as BLOCK_X and BLOCK_Y, and CTA-specific parameters such as blockId.X and blockId.Y and thread-specific parameters within each CTA. In the example, values computed from CTA-specific parameters are represented as C_1 and C_2. The pitch value, C_3, is essentially the parameter value which is constant across all threads in the kernel. Thus each CTA needs to compute its own C_1 and C_2 values first to compute the base address which is represented as $\theta = C_1 + C_2 \times C_3$ in these examples. Once a CTA's base address is computed each thread can then use its thread id (represented by threadId.x and threadId.y) and the stride value represented by C_3 to compute the effective array index.



(a) LPS (b) BFS

Figure 6: Load address calculation example (LPS [6] and BFS [8])

For example, the CTA of LPS consists of a (32, 4) two dimensional thread group. Given that a warp consists of 32 threads, each CTA has four warps. The threads in the same SIMT lane position in all four warps have the same thread x dimension id (from 0 to 31), and the y dimension id distance between consecutive warps is one. Therefore, the load distance between two consecutive warps within each CTA is a fixed value, represented by the C_3 in the equation. This distance can be easily calculated at runtime by subtracting the load addresses of any two consecutive warps in the same CTA. This distance then can be used across all the CTAs. However, the CTA-specific constant values C_1 and C_2 must be computed for each CTA separately.

Note that the base address of CTA is cannot be predicted easily even if it appears to be a function of CTA id. Because this function itself varies from one load to another load instruction in the same kernel, and differs across kernels in the benchmark, and most certainly differs across benchmarks as is clear from the LPS and BFS computations. Also inter-CTA distances (difference of base addresses between two CTAs) in a SM is irregular. For example, CTAs (0,0), (3,3), (7,2) and (11,1) are all initiated in the same SM for LPS in our simulation run. Thus the example load shown in the LPS figure when executed in the same warp ids across different CTAs do not exhibit any stride behavior across CTAs. For instance, the distance between the load address executed in warp0 in CTA(0,0) and CTA(3,3) is 5184, while the distance between the same load in CTA(3,3) and CTA(7,2) is 6272.

Based on these observations, the prefetch address of all the warps within each CTA can be calculated only once the base address and stride values are computed. The stride value can be computed by subtracting the load addresses of two consecutive warps (based on warp-ids) within the CTA for the load. But the base address must be computed first by at least one warp associated with each CTA.

Across a range of GPU applications we evaluated, the stride value in fact can be computed from two consecutive warps within the CTA. One exception is the indirect references that graph applications normally use to find neighboring node and edges as shown in Figure 6b. g_graph_edges, $g\_graph\_visited$, $g\_cost[id]$ and $g\_updating\_graph\_mask$ are indexed by variable i which is a value loaded from $g\_graph\_nodes[tid]$. Therefore, the address of these indirectly referenced variables cannot be predicted using stride prefetcher. However, the metadata addresses ($g\_graph\_mask[tid]$, $g\_graph\_nodes[tid]$ and $g\_cost[tid]$) are all thread-specific references and these addresses can be calculated using thread id and CTA id as illustrated in the blue box.

4. CTA-aware Prefetch

CTA-aware prefetcher (CTAA) exploits the regularity of memory access patterns among warps within a CTA, similar to inter-warp stride prefetching. However, unlike inter-warp stride prefetching, CTAA detects base address changes across CTA boundaries to increase the accuracy of prefetching.

4.1 Scheduling Algorithm Overview

We first describe our prefetching approach at the algorithmic level and then provide the necessary architectural support details later.

CTAA tracks the base address for a given CTA, say CTA_{trail}, by executing one warp, say W_{lead}, early in another CTA, CTA_{lead}. We call the warp that computes the base address of a given **CTA$_{trail}$** as the leading warp, and **CTA$_{trail}$** itself is called the trailing CTA. The **CTA$_{lead}$** where the leading warp is executed early is called the leading CTA. Thus for every CTA there is one leading warp that is executed first in a leading CTA to compute the prefetch base address.

The conventional two-level scheduler initially enqueues warps from each CTA to the ready queue in CTA order; warps of the first CTA are first enqueued to the ready queue and then the warps of the following CTAs are enqueued until the ready queue is filled up. In order to detect the base address and stride information as early as possible, the two-level warp scheduler [13] is modified. The CTAA scheduler initially picks one CTA arbitrarily, amongst all the assigned CTAs to the SM, as the **CTA$_{lead}$**. It then selects one warp **W$_{lead}$** from each CTA that is currently scheduled for execution on the SM, and then places them in the ready queue. The reason for selecting the leading warps to be placed first in the ready queue is to enable early calculation of CTA base addresses. Once all the **W$_{lead}$**s for all CTAs are placed in the ready queue then the CTAA scheduler places as many of the remaining warps from the **CTA$_{lead}$** into the ready queue to fill up the ready queue. These warps are called trailing warps (W_{trail}) and used for the stride calculation, which will be explained shortly.

The second necessary component is the calculation of the stride. Stride information can be extracted by calculating the distance between two successive warps in the same CTA. Once **W$_{lead}$** of a CTA computes the base address of the load instruction, the stride distance can be calculated when one of the W_{trail}s of the same CTA is scheduled and computes its own load address for the same load instruction. In many applications, CTAs typically use the same stride for the same load instruction. Therefore, the stride that is computed by the **W$_{lead}$** and **W$_{trail}$** of **CTA$_{lead}$** is used for calculating the prefetch addresses of the warps in **CTA$_{trail}$**s.

Figure 7 is the simplified illustration of the modified scheduling order. Each rectangle of the figure indicates a warp. Assume that a CTA consists of four warps and there are three CTAs. Suppose that

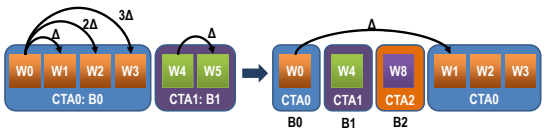


Figure 7: CTAA scheduling order: One leading warp of each trailing CTA is scheduled alongside a leading CTA to compute the base address and stride

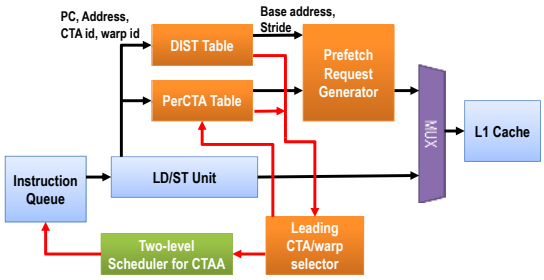


Figure 8: Hardware structure of CTA-aware prefetcher

the ready queue size is six. The figure shows the ordering of the six warps in the ready queue. The left hand side figure shows the order in which warps are placed in the ready queue using conventional two-level scheduling algorithm. In this priority ordering, prefetch requests for W5, W6 and W7 cannot be issued until W4 computes the base address, B1, for CTA1. Furthermore, prefetch addresses for warps from CTA2 cannot be even initiated until at least one of the warps from CTA2 is promoted into the ready queue. The right hand side figure represents the modified CTAA scheduler where leading warps of CTA0, CTA1 and CTA2 are scheduled with higher priority followed by trailing warps (W1, W2 and W3) of the leading CTA (CTA0). Prefetch addresses (B1+Δ and B2+Δ) for trailing warps of the trailing CTAs (CTA1 and CTA2) can be calculated right after the Δ is computed from W0 and W1 in the leading CTA (CTA0).

We describe how prefetches are launched and inaccuracies are controlled using the base address and stride values in Section 4.2.

4.2 Microarchitectural Support for CTAA Scheduling and Prefetching

Figure 8 shows hardware structure of our CTAA scheduler and prefetcher. To detect the base address and stride values, two structures are added: DIST table and PerCTA table. A prefetch request generator is added, which is a simple adder logic block that issues a prefetch instruction.

Table	Fields	Total bytes
DIST	PC (4B), stride (4B), mispredict counter (1B)	9B
PerCTA	PC (4B), leading warp id (1B), base address (4×4B)	21B

Table 2: Database entry size of prefetch engine

PerCTA table: The purpose of the PerCTA table is to store the base address of a targeted load from each CTA using the early base address computation of a leading warp. Since each CTA has its own base address it is necessary to store this information on a per CTA basis. Even though each leading warp in a CTA has 32 threads and hence can potentially compute 32 distinct base addresses (one per each thread) our empirical evaluations showed that prefetching is ineffective when the load instruction generates many uncoalesced memory accesses. Thus we only target those loads that generate no more than four coalesced memory accesses. A single 4×4 byte base address vector is used to store the base address of a targeted load within each CTA.

In our design we only target prefetching at most two distinct loads (identified by their program counters) within each CTA. Hence, the PerCTA table has two entries. Each entry of PerCTA table stores the load PC, leading warp id, and base addresses. When a warp executes a load, the PC is used to search the two entries in the corresponding PerCTA table. If the load PC is not found in the table then it indicates that no warp in that CTA has reached that load PC and hence the current warp is considered as the leading warp for that load instruction. Then the leading warp id, load PC, and the access addresses from that warp are stored in the PerCTA table. Since the PerCTA table has two entries, if there is no available entry in the PerCTA table, the least recently updated entry is evicted and the new information is registered in that entry. But in most of our benchmarks the targeted prefetch loads are two to four load instructions and hence this replacement policy did not significantly alter the performance.

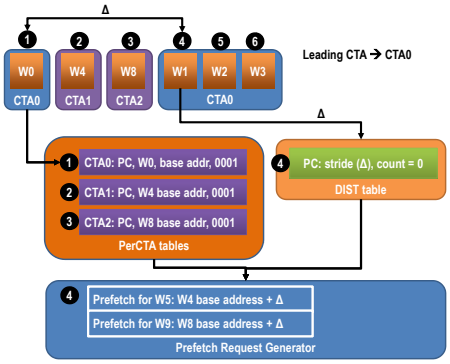
DIST table: An entry in the DIST table contains the load PC, stride value and a misprediction counter. Unlike PerCTA table, DIST table is shared by all CTAs since stride value is shared across all warps and across all CTAs. Each entry of DIST table is associated with a load instruction indexed by the load PC. When a load instruction is issued, the DIST table is accessed alongside the PerCTA table with the load PC. If no matching entry is found in the DIST table while an associative entry is found in the PerCTA table then it indicates that the base address of the CTA is already calculated while the stride value is not. Therefore, the stride needs to be calculated by using the stored base address and the current warp's load address. Note that the stride computation between two warps can generate potentially four different values across maximum of four distinct memory requests for the same load instruction. If the stride value for all memory requests between the two warps are not identical then we simply assume that the PC is not a striding load and the PerCTA entry for that PC is invalidated by setting the PC bits to zero. If on the other hand the stride computation returns just one value then that stride value is stored in the DIST table. We also set the misprediction counter to zero at that time.

If the load PC is found in the DIST table, then we already have the stride value stored in the DIST table. The misprediction counter is then accessed. If the misprediction counter is larger than a threshold then no prefetch is issued to prevent inaccurate prefetches. Otherwise, prefetches are generated as described in the next section. The misprediction counter is two bytes and the threshold is set to 128 by default.

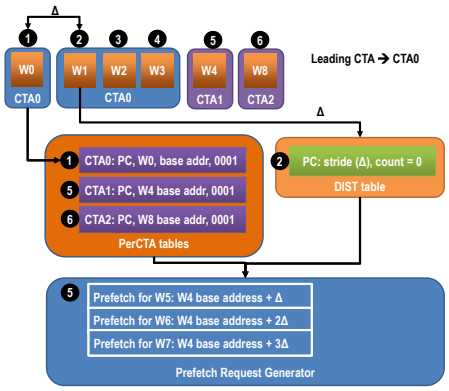
To throttle inaccurate prefetches, the address of each prefetch is verified by comparing with the address of the actual demand fetch. Thus every warp instruction that issues a demand fetch also calculates the prefetch address to detect a misprediction. The misprediction counter increases by one whenever the calculated prefetch memory address is not equivalent to the demand fetch.

4.3 A Simple Prefetch Generation Illustration

We illustrate the entire prefetch algorithm with a simple illustration showing how prefetches are issued. Prefetches are triggered under two different scenarios. In the first case, prefetch requests are generated when trailing warps of the leading CTA execute a load instruction after the base addresses of the CTAs are registered to the PerCTA table by their leading warps. This case is illustrated in Figure 9a. The number in the circle above each warp id indicates the order of each warp's load instruction issue. In this illustration assumes that W0, W4, and W8 have already finished execution and they have updated the PerCTA table. But there is no stride value that



(a) Case 1: base addresses are settled before stride detection



(b) Case 2: stride is detected before base addresses are settled

Figure 9: Cases for prefetch request generation

is stored in the DIST table as yet since none of the trailing warps has been executed. When W1, which is a trailing warp of CTA0, issues the load instruction the stride (Δ) value is computed. Then the prefetcher traverses each of the PerCTA tables with the PC value of W1. Whenever the PC is matched in a given PerCTA table, the base addresses are read from the table and then new prefetches are issued for the matched CTA. In this example as base addresses and stride are ready for CTA1 and CTA2, prefetch requests for W5 and W9 are generated.

The second scenario for prefetching occurs when the stride value is calculated before the base addresses of the trailing CTAs are registered to the PerCTA table. This happens when the leading warps of trailing CTAs are scheduled behind the trailing warps of the leading CTA. In spite of the best effort by the scheduler to prioritize all the leading warps to the front of the ready queue, it is possible that some of the trailing warps of leading CTA are executed ahead of the leading warps of trailing CTAs. Figure 9b shows an example of this case. In this example W1 executes ahead of W4 and W8. As W1 of CTA0 issues a load instruction before W4 and W8 of CTA1 and CTA2, then Δ is computed and stored in DIST table before the PerCTA table is updated with base addresses for CTA1 and CTA2. When W4 is issued, after updating PerCTA table with the base address as in the first scenario, W4 also generates prefetch requests for other warps in CTA1 by using the stride value that is already computed in DIST table. Thus in this scenario the leading warp of trailing CTA issues prefetches for all the trailing warps of its own CTA.

4.4 Warp Wakeup on Data Arrival Optimization

To avoid prefetched data from evicted before consumption, the warps are woken up when the data arrives. If the warp is already in the ready queue, nothing happens. Otherwise, the warp is moved to the ready queue eagerly by pushing a ready warp forcibly into the pending queue. Similar approach was proposed by OWL [16]. Only minimal change is needed for implementing the eager warp wakeup. When a warp sends a load request to L1 cache, the warp id is bound with the request so that the returned value is sent to the right warp. For the warp wakeup, the id of the warp that will be fed by the prefetched data is bound to the memory request. When the data arrives, warp scheduler is requested to promote the warp that is bound to the prefetch memory request to the ready queue.

5. Evaluation

5.1 Settings and workloads

We implemented the CTAA scheduler and prefetcher on *GPGPU-Sim v3.2.2* [6]. The baseline configuration is similar to Fermi (GTX480) [27]. Each SM has a 16KB private L1 data cache. The shared L2 cache for all 15 SMs is 768KB. The global memory is partitioned into 6 DRAM channels where each DRAM channel has a FR-FCFS (First-Ready, First-Come-First-Served) memory controller and each channel is associated with two sub-partitions of 64KB unified L2 cache. Timing parameters of the global memory is set based on GDDR5 with 924MHz memory clock [15]. Detailed configuration parameters is listed in Table 3.

We used 16 benchmarks selected from different GPU benchmark suites as listed in Table 4. All applications were simulated until the end of their execution or when the simulated instruction count reached one billion instructions. CTAA performance is compared to the baseline architecture using two-level warp scheduler with the ready warp queue size of 8 entries. Additionally, several previously proposed GPU prefetching methods are implemented to compare the relative performance benefits of CTAA.

Parameter	Value
Simulator	GPGPU-Sim v3.2.2
Core	1400MHz, 32 SIMT width, 15 cores
Resources / core	48 concurrent warps, 8 current CTAs
Register file	128KB
Shared memory	48KB
Scheduler	two-level scheduler (8 ready warps)
L1I cache	2KB, 128B line, 4-way
L1D cache	16KB, 128B line, 4-way, LRU, 32 MSHR entries
L2 unified cache	64KB per partition (12 partitions), 128B line, 8-way, LRU, 32 MSHR entries
DRAM	924MHz, ×4 interface, 6 channels, FR-FCFS scheduler, 16 scheduler queue entries
GDDR5 Timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6, t_{CDLR}=5, t_{WR}=12 [15]

Table 3: GPGPU configuration

Benchmark	Abbr.	Benchmark	Abbr.
lapalce3D [6]	LPS	histogram [26]	HST
backprop [8]	BPR	MonteCarlo [26]	MC
hotspot [8]	HSP	jacobi1D [2]	JC1
pathfinder [8]	PTH	FFT [3]	FFT
mri-q [1]	MRQ	MD [3]	MD
sgemm [1]	SGE	Breadth First Search [8]	BFS
stencil [1]	STE	Kmeans [14]	KMN
convolutionSeparable [26]	CNV	MatrixMul [26]	MM

Table 4: Workloads

5.2 Performance enhancement

Figure 10 shows the reduction in execution cycles of prefetching methods normalized to the baseline configuration using two-level

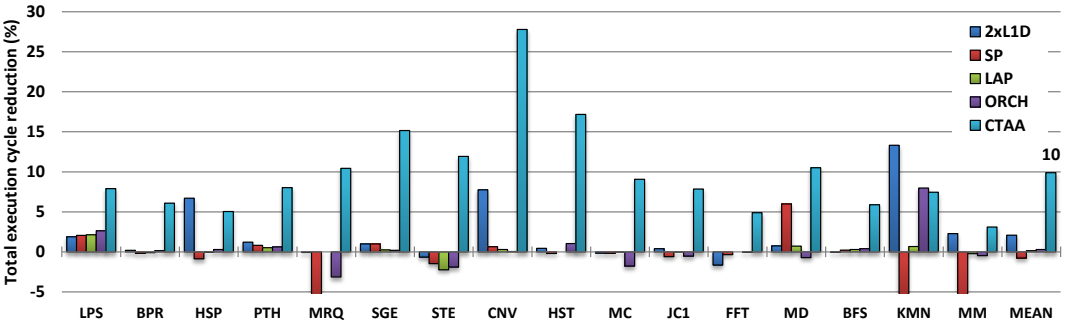


Figure 10: Speedup of prefetcher over two level scheduler without prefetch

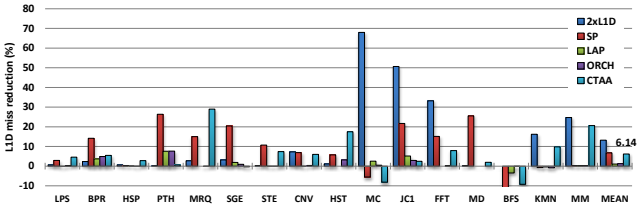


Figure 11: L1D cache miss reduction

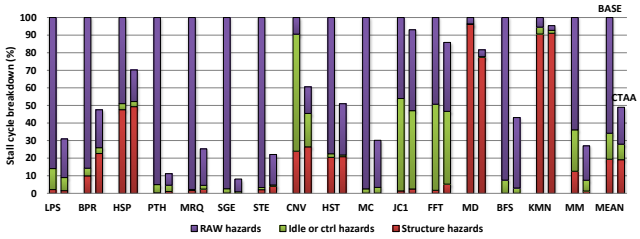


Figure 12: Breakdown of reasons of pipeline stalls

warp scheduler without prefetching. Data labeled CTAA is the CTA-aware prefetching, LAP is the locality-aware prefetching built on top of two-level scheduler, where a macro block of 4 cache lines is prefetched if more than or equal to two cache lines are missed within each macro block of L1 data cache [17]. ORCH is the orchestrated prefetching where LAP is further enhanced with the prefetch-aware warp scheduling as described in [17]. SP means the simple next-line prefetcher which prefetches next cache line if one cache line is missed. 2xL1D is the baseline configuration with two-level warp scheduler with twice the L1 cache size but with no prefetching.

Figure 10 shows CTAA reduces execution time by up to 28%, and 10% on average. CTAA performs better than doubling the L1 cache size (2xL1D) on average. In fact doubling the cache size has about 2% performance improvement. The reason for this small improvement in performance is the lack of strong temporal locality in the data. ORCH improves performance by about 1% and it is lower than the performance improvements reported in [17]; the primary reason is that in the original work the prefetcher is implemented on different baseline settings - 30 cores with GDDR3 DRAM whereas our CTAA is tested on more recent Fermi architecture.

Figure 11 shows that cache miss ratio is decreased for most benchmarks with CTAA. On average, L1 data cache miss ratio is reduced by 6.14%, which is better than ORCH and LAP. When ORCH and LAP were implemented on two-level scheduler baseline both of them reduced the L1 cache miss rate by just 1%. Surprisingly, miss ratio reduction of CTAA is slightly worse than SP, which reduced the miss rate by 6.73%. Note that a cache hit is counted on a tag match even if the data is still being transferred from the next level in the memory hierarchy in response to a prefetch or a prior demand request to that cache line. Thus the reason for the discrepancy between miss ratio and execution time is the pipeline stall reduction and prefetch accuracy and coverage shown next.

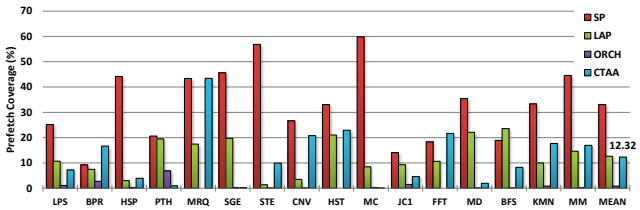
5.3 Pipeline stall reduction

We monitored pipeline stalls while running CTAA and analyzed how much of the pipeline stalls are reduced or increased compared to the baseline two-level scheduler without prefetch. Note that a pipeline could be stalled due to three main reasons: RAW hazards, control hazards, and structure hazards. RAW pipeline stall occurs when operands of an instruction are not ready. There are two reasons that cause the RAW pipeline stall: memory access operation and data dependencies on a prior long latency ALU operation, such as an SFU operation. CTAA primarily targets memory access delays and reduces these stall times significantly. However, if too many prefetch requests are generated in a short time window, prefetch requests may consume hardware resources, such as MSHRs in the data cache, which then increases the structural hazards. Idle and control hazards are closely related to warp scheduling order. Control hazards can also be reduced if the control instruction itself is dependent on a load operation. If the load latency is reduced with prefetch CTAA will also indirectly reduce control hazards as well.

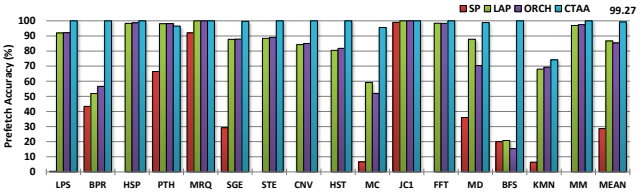
Figure 12 shows the breakdown of the three main pipeline stall reasons. Among the two bar charts for each application the left hand side bar is the pipeline stall breakdown measured while running two-level scheduler without prefetch. The right-hand side bar is the pipeline stall breakdown with CTAA normalized to the pipeline stalls without prefetch for each of the stall categories. On average total pipeline stalls are reduced by 50% with CTAA. RAW and control stall cycles are decreased by 67% and 36% respectively, while the wasted cycles due to structure hazards stay the same. Reducing the stall cycles does not translate into a correspondingly large reduction in the execution time. The reason for this discrepancy is that part of the stall time can be hidden by other warps.

5.4 Coverage and accuracy of prefetching

Figure 13 shows the coverage and accuracy of CTAA. Coverage is defined as the ratio of number of memory requests that a prefetch is issued compared to the total demand fetch requests. CTAA on

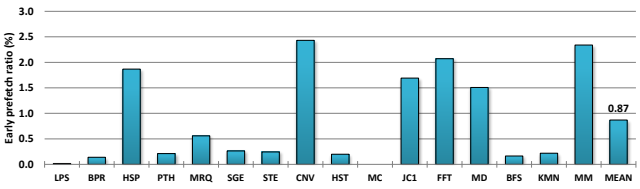


(a) Coverage

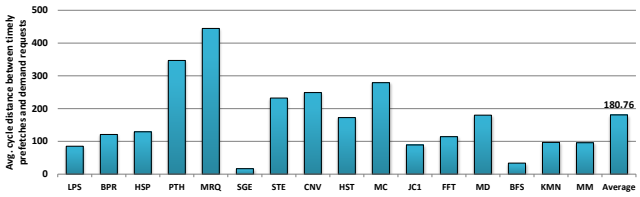


(b) Accuracy

Figure 13: Prefetch coverage and accuracy

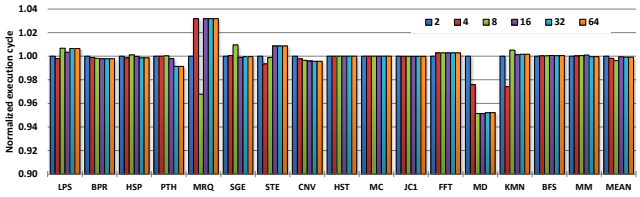


(a) Early prefetch ratio

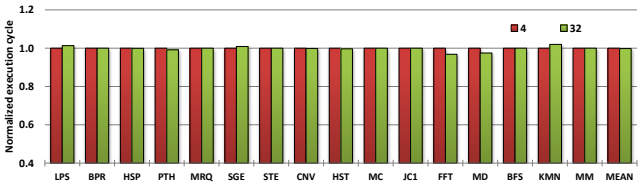


(b) Prefetch distance of timely prefetches

Figure 14: Early prefetch ratio and average prefetch distance of timely prefetches



(a) Performance impact of # entries of PerCTA table and DIST table



(b) Performance impact of # base addresses per instruction

Figure 15: Performance variation w.r.t. # entries of PerCTA table and DIST table and # base addresses per load instruction

average provides 12.19% coverage. Note that coverage is lower for benchmarks such as PTH, HSP and BFS that traverse graphs. In these benchmarks strides are rare. One advantage of CTAA is that it recognizes the lack of strides and then avoids issuing wasteful prefetch requests, thereby curtailing coverage. Higher coverage ratio doesn't always improve performance, as shown by the result of the SP, since inaccurate prefetches only consume resources like cache space and data bandwidth.

Accuracy is the ratio of prefetch requests that were actually consumed by the demand requests. Accuracy is an important factor because unnecessarily prefetching data increases bandwidth and causes cache pollution. As shown in Figure 13b, the accuracy of the CTAA is near 100% for most benchmarks. In the worst case the accuracy is around 70% for graph applications, such as KMN, that have complex access patterns. But note that for the other graph applications such as PTH, HSP and BFS, CTAA quickly recognizes the lack of strides and shuts down prefetching. Thus the coverage is quite small but the accuracy is over 95%. In BFS, the search process streams through the graph nodes in a predictable manner at each level (or breadth) of the graph. Thus CTAA is able to occasionally capture the strides during breadth first traversal, leading to slightly higher coverage (8.24%) with higher (100%) accuracy than PTH. In case of KMN, the application goes through longer periods of predictable data access patterns, thereby triggering the CTAA to prefetch, but just as the CTAA prefetcher settles into a steady state, the stride pattern is interleaved with unpredictable traversal patterns. Thus the prefetch coverage for KMN is nearly 18% but the accuracy is under 80%.

5.5 Timeliness of prefetching

When prefetch is issued too early, the prefetched data can be evicted before the actual load is issued due to the limited L1 cache capacity. Such early prefetch only increases memory traffic without benefit. As stated in Section 4, CTAA adjusts warp priority to detect the stride and base address of CTAs as early as possible and to increase the distance between prefetch and demand requests. Additionally, a warp in the pending queue is awakened when the corresponding data prefetch reaches L1 data cache. Hence, CTAA can adjust prefetch timing for target load instructions effectively to improve performance. Figure 14a shows the percentage of prefetched data that was evicted before use. On average only 0.87% of the prefetched data was evicted from L1 by an intervening demand fetch before the prefetched data was consumed. Therefore, almost all the prefetches issued by CTAA are effectively consumed by destination warps.

On the other hand, if the distance between prefetching and demand requests is too short, prefetcher cannot effectively hide the long latency of memory operation. Given that latency of memory operation of GPUs is hundreds of cycles, prefetch requests should be issued sufficiently far ahead before demand requests are issued. Figure 14b shows the distance between prefetch and demand requests when CTAA is applied. On average, CTAA issues a prefetching request about 180.76 cycles before the targeted demand request.

5.6 Hardware overhead

CTAA uses two tables: DIST and PerCTA. One DIST table per SM, one PerCTA table per CTA. Both tables are accessed by a load instruction. By default, there are two entries per PerCTA and two entries per DIST table. In Fermi architecture, each SM can run at most eight CTAs. Therefore, the area overhead per SM for these two tables is 354 Bytes ($8\ CTAs\ per\ SM \times 21B\ PerCTA\ entry \times 2\ PerCTA\ entries + 9B\ DIST\ table\ entry \times 2\ DIST\ entries$).

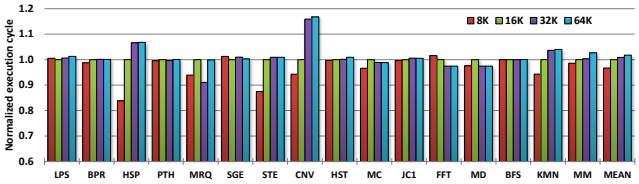


Figure 16: Performance variation w.r.t. L1D cache size

We measured the performance variation while varying the size of PerCTA table. Figure 15a shows the normalized execution cycle when each of the PerCTA table and DIST table is changed to 2, 4, 8, 16, 32, and 64 entries. On average, only negligible performance increase was observed when using larger PerCTA tables. There were only two exception cases; MRQ derived 3% performance gain and MD had 5% performance degradation when a table has more than 16 entries as it was able to capture many more striding loads in these two benchmarks.

As explained earlier, CTAA does not target loads that issue more than four un-coalesced memory accesses for prefetching. To quantify the impact of this restriction we measured the performance when CTAA does target loads with many more un-coalesced memory accesses. Figure 15b shows the normalized execution cycles when we target loads that can generate up to 32 base addresses (one base address for each of the 32 threads in a warp). The performance decreases on average when generating prefetches for the un-coalesced memory operations. As mentioned earlier, when memory requests are un-coalesced issuing many concurrent prefetches leads to increased resource contention.

5.7 Cache size sensitivity

Cache size plays an important role in prefetch performance. If cache size is insufficient to accommodate the prefetched data, prefetch data will be replaced before their use. We evaluated the cache size impact on prefetch performance while varying L1D size from 8KB to 64KB. We preserved the shared memory size to be 48KB in this evaluation. Figure 16 shows the execution cycle normalized to the default configuration (16KB L1D). On average, increasing the cache size to 64KB increases the performance by only 2% compared to using just a 16KB cache with prefetching. As mentioned earlier, the reason for this small performance improvement even after using a larger cache size is the lack of temporal locality. Furthermore, as shown in Figure 14a, very few prefetches were displaced from cache before they were used in the default configuration. Hence there was no need for larger cache size to hold the prefetched data.

6. Related Work

Prefetching is one of the prominent ways to overcome the memory stalls. Hardware and software prefetching approaches for data or instruction fetching have been studied and applied to modern microprocessors. An excellent survey of prior prefetching mechanism is available in [35].

Many prefetching techniques [4, 5, 9, 11, 22, 23, 30, 34] were developed in the context of SMPs and CMPs. We focus primarily on prior GPU prefetching and prior stride prefetching approaches since that is the focus of this work. The simple stride prefetching, described in [30], was proposed in the context of CPUs. Baer and Chen [5] extended sequential prefetching to data arrays having variable strides. The address of prefetching request is predicted from previous address and stride information indexed by a PC of a load instruction. When applied in the context of GPUs with thousands of threads the base address and stride values are obfuscated due to CTA and warp scheduling approaches, which is the concern we tackle in our research.

Effectiveness of prefetching has been studied because inaccurate prefetching requests may generate unnecessary memory traffic to increase latency and waste resources in memory hierarchy [10, 32, 36]. Jouppi [18] proposed to add additional buffers to store prefetched data to prevent cache pollution. Srinath et al. [31] presented the mechanism to control aggressiveness of prefetching by monitoring cache pollution caused by prefetching as well as accuracy and timing of prefetching requests. By controlling frequency of prefetching, they reduce side effects of prefetching. CTAA also uses similar throttling mechanisms to control wasteful prefetching.

Several studies proposed memory prefetching algorithms for GPU [17, 20, 21]. Lee et al. [21] proposed a software and hardware based many-thread aware prefetching which basically commands threads to prefetch data for the other threads. They exploit the fact that the memory addresses are referenced using thread id in many GPU applications. Hence, by computing the stride value from successive thread ids, they prefetch one thread's data using one of prior threads' load addresses as the base address. The simple stride prefetching across warps works well within a CTA. But as shown in this work the number of warps per CTA is limited. Each CTA's base address is not predictable from a prior CTA's base address and the complex GPU scheduling algorithms, other than round-robin, make it difficult to detect strides even within a CTA. These are tackled by the CTAA prefetcher effectively.

Jog et al. [17] proposed a new prefetch-aware scheduling policy which schedules consecutive warps in different scheduling group so that the warps in a scheduling group can prefetch data for the logically consecutive warps that are scheduled in different scheduling groups. By distributing consecutive warps that are likely to access near addresses, the proposed scheduling algorithm also derives better bank level parallelism. We compare CTAA quantitatively with this approach and showed the performance improvements of CTAA.

Lakshminarayana and Kim [20] proposed a prefetching algorithm by observing an unique data access patterns in graph applications. Unlike other studies, they prefetch data to the spare register file. Whereas their work focused on prefetch for iterated load instructions in a loop that appears mostly in graph applications for GPUs, CTAA is effectively applicable to load instructions regardless of number of iterations as far as load instructions have regular stride across warps. Evaluation results show that performance is benefited for graph applications as well as generic GPGPU programs by CTAA

7. Conclusion

Due to the nature of computations GPU applications exhibit stride access patterns. But the starting address of a stride access is a complex function of the CTA id and thread id and other application defined parameters. Hence, the base address of stride varies from one CTA to another. Furthermore, GPU scheduling policies try to balance the workload across all available computational resources. As a result, contiguous thread blocks are split over different SMs in a GPU and even warps within a thread block have unpredictable execution schedule. The combined effect of all these issues is that even well defined stride accesses are obfuscated in the GPU execution stream. To tackle this challenge we propose CTA-aware prefetcher (CTAA) for GPUs. CTAA hoists the computation of the base address of each CTA by scheduling one leading warp from each trailing CTA to execute alongside the warps of a current leading CTA. The leading warps compute the base address for each trailing CTA, while the stride value is detected from the execution of trailing warps of the leading CTA. Using the per-CTA base address and combining with the global stride value that is shared

across all CTAs, the proposed prefetcher is able to issue timely and accurate prefetches. Using simple counter based throttling mechanisms the prefetcher can control wasteful prefetches when strides are not present in an application. By reordering the warps' scheduling order, the prefetch distance is increased. The evaluation results show that the CTAA predicts prefetch addresses with over 99.27% accuracy and improves performance by 10% on average.

References

[1] Parboil benchmark suite. URL `http://impact.crhc.illinois.edu/parboil.php`.

[2] Polybench/gpu. URL `http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/`.

[3] Scalable heterogeneous computing benchmark suite. URL `http://keeneland.gatech.edu/software/keeneland/shoc`.

[4] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *ISCA*, pages 52–61, 2001.

[5] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *SC*, pages 176–186, 1991.

[6] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, pages 163–174, 2009.

[7] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing dram latency divergence in irregular gpgpu applications. In *SC*, pages 128–139, 2014.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[9] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-m. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *MICRO*, pages 69–73, 1991.

[10] F. Dahlgren and P. Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *HPCA*, pages 68–77, 1995.

[11] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *TPDS*, 6(7):733–746, 1995.

[12] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *MICRO*, pages 102–110, 1992.

[13] 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 *ISCA*, pages 235–246, 2011.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *PACT*, pages 260–269, 2008.

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

[16] 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. volume 41, pages 395–406, 2013.

[17] 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 *ISCA*, pages 332–343, 2013.

[18] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *HPCA*, pages 28–31, 1990.

[19] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *PACT*, pages 157–166, 2013.

[20] N. B. Lakshminarayana and H. Kim. Spare register aware prefetching for graph algorithms on gpus. In *HPCA*, 2014.

[21] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *MICRO*, pages 213–224, 2010.

[22] Y. Lie and D. Kaeli. Branch-directed and stirde-based data cache prefetching. In *ICCD*, pages 225–230, 1996.

[23] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *JPDC*, 12 (2):87–106, 1991.

[24] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue - GPU Computing*, 6(2):45–53, 2008.

[25] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`, .

[26] NVIDIA. NVIDIA CUDA SDK 2.3. `http://developer.nvidia.com/cuda-toolkit-23-downloads`, .

[27] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, .

[28] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *MICRO*, pages 72–83, 2012.

[29] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *PACT*, pages 73–82, 2013.

[30] A. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, Dec 1978.

[31] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, pages 63–74, 2007.

[32] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A prefetch taxonomy. *TC*, 53(2):126–140, 2004.

[33] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *SC*, 2009.

[34] S. P. Vanderwiel and D. J. Lilja. When caches aren't enough: Data prefetching techniques. *Computer*, 30(7):23–30, 1997.

[35] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *CSUR*, 32(2):174–199, 2000.

[36] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr., and J. Emer. Pacman: Prefetch-aware cache management for high performance caching. In *MICRO*, pages 442–453, 2011.

[37] G. L. Yuan, A. Bakhoda, and A. T. M. Complexity effective memory access scheduling for many-core accelerator architecture. In *MICRO*, pages 34–44, 2009.