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

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

Abstract
Several studies have been proposed to adopt memory prefetching schemes to reduce performance impact of long latency memory operations in GPUs. By leveraging a simple intuition that the consecutive warps are likely to have spatial locality, prior approaches prefetch two or four consecutive cache lines when there is a cache miss. Other approaches predict striding accesses by detecting base address and stride value from each warp’s load address history. Warp-based load prediction works well when a load instruction is repeatedly executed in a loop. But to exploit parallelism, GPU kernels favor creating a massive number of threads rather than sequential loop codes. In this paper, we exploit the observation that all the threads generated from a kernel execute the same code segment. When threads are grouped into cooperative thread arrays (CTAs), each thread uses thread id and CTA id to identify the data that it operates on. Thus the stride values for loads among warps within a cooperative thread array (CTA) can be easily detected and this stride value can be applied across all the CTAs in the kernel. However, the starting base address accessed by the first warp in a CTA is difficult to predict since that starting address depends on how the CTAs are created by the application programmer. Hence, we propose to first compute the base address of a load in each CTA by using a leading warp. The leading warp of each CTA is executed early by pairing it with warps from currently executing CTA. Thus our proposed CTA-aware prefetch predicts all the trailing warps’ load addresses by first computing the base address early through a leading warp and then using the stride value. 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 94% accuracy and is able to improve performance by 5.4%.

1. Introduction
Long latency memory operation is one of the most critical performance hurdle in any computation. Graphics processing unit (GPU) relies on dozens of concurrent warps for hiding the performance overhead of long latency operation by quickly context switching among all the available warps. When a warp issues a load instruction and it encounters a cache miss, one of the other ready warps takes their turn and resume their execution. 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 [7, 16] employ two warp queues: pending queue and ready queue. Warps in the ready queue are scheduled for execution and when a warp in the ready queue encounters a long latency operation it is pushed out into the pending queue and a 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. 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 [15, 11, 10, 20, 13] can be categorized into three categories: intra-warp stride prefetching, inter-warp stride prefetching and next line prefetching.

1.1. GPU architecture

Before describing the applicability and limitations of these three approaches 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 based on NVIDIA Fermi architecture [19]. Each SM has 32 single instruction multiple thread (SIMT) lanes where each SIMT lane has its own execution units. Each SIMT lane is treated as one cuda core. Also, an SM is associated with its own private memory subsystem, register files and level-1 texture, constant, data and instruction caches. SMs are connected to level-2 cache partitions that are shareable to all SMs via an interconnection network. For larger data bandwidth and mem-

Figure 1: GPGPU hardware and software architecture
ory level parallelism, multiple external DRAM chips, called global memory are directly connected with a GPU device through memory controllers where each controller is associated with one or more level-2 cache partition. If requested data is serviced by external DRAM chip, the latency, which varies with memory traffic congestion of DRAM, is hundreds or even thousands of core cycles [25].

The GPU software execution model is shown in Figure 1b. Normally, a GPU application is composed of many kernels, which are basic function modules to execute massive number of threads. 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 a streaming multiprocessor (SM) in a GPU. Threads in a CTA is sub-grouped into a warp - the smallest execution unit that sharing the same program counter across all threads within the warp. As our baseline hardware has 32 SIMT lanes, a warp contains 32 threads. For memory operation, a memory request is generated for each thread and multiple requests are merged if 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, shared memory usage – the available resources within an SM must be exceed or the CTA resource demands. 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 a maximum of 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.

Figure 2 shows an example CTA distribution where each SM can run two concurrent CTAs and the kernel consists of 12 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. Note that two consecutive CTAs are not assigned to an SM for load balancing; suppose that there are fewer CTAs than the number of SMs, then the best way to balance the load is to assign one CTA per SM first, even though each SM can accommodate two CTAs. Once the six CTAs are first allocated to all the SMs, the 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. Thus the second round of CTA assignments is determined dynamically based on CTA termination order.

1.3. Limitations of prefetches in GPUs

With the above background, we now describe the limitations of GPU prefetchers and how one can overcome these challenges. **Intra-warp stride prefetching:** Prefetching of strided data requests is a basic prefetching method that is effective when array data is accessed with regular indices in a loop [6, 3]. In the context of a GPU application if a warp loads array data from memory repeatedly in a loop then stride prefetching is initiated for prefetching data for future loop iterations. The load operation is identified by the program counter (PC) and regular address offsets are detected between memory requests of different time stamps. Since each prefetch targets the load instruction of a future loop iteration within the same warp, this approach is called *intra-warp stride prefetching*. Intra-warp stride prefetching was recently proposed for graph algorithms running on GPUs [13].

The effectiveness of intra-warp stride prefetching depends on the presence of loops to access regular data structures. But there is a growing trend towards replacing loop operations in a GPU applications with parallel thread operations. This observation has also been made in a prior study that showed that deep loop operations are seldom found in GPU applications [17, 18]. When a loop intensive C program is ported to CUDA (or openCL), loops are reduced (or even eliminated in many cases) to leverage massive thread level parallelism. Figure 3 shows a simple illustration of the C and CUDA code of a vector addition program. To add two N-way vectors, C program runs a loop of N iterations. On the other hand, in GPU that can run hundreds of threads concurrently, the loop is replaced by a line of vector addition code. Instead of running...
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.

As shown in Figure 2, SMs are not assigned consecutive CTAs. Without re-writing the code.

Thus within a CTA all the warps are able to see stride accesses to the same CTA. But the prefetcher is unable to prefetch across CTAs assigned to the same CTA. Figure 4 shows an example of disruptions to regular striding between warp 2 and warp 3 is not Δ because they straddle CTA boundaries. With a simple inter-warp stride prefetching wrong prefetching address of warp 3 is issued. Note that a well designed stride prefetcher detects this change and eventually corrects the new base address and stride for CTA 3. But during this transition either prefetching is suspended or in the worst case wrong prefetches may be issued. Furthermore, every time a correction is made to the stride prefetcher the timeliness of prefetching is compromised. Thus a prefetch for warp 5 (W5) cannot be issued until the stride prefetcher is able to detect the new base address.

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. Next line prefetching in conjunction with warp scheduling policies for GPUs was proposed in [11, 10]. The proposed warp scheduler assigns consecutive warps in different scheduling groups so that the warps in a scheduling group can prefetch data for the logically consecutive warps and timeliness of prefetching is ensured. If the logically consecutive warps have high spatial locality, high accuracy of the prefetch requests is guaranteed for upcoming memory accesses of following warps.

However, our analysis on general purpose applications that are ported to run on GPUs shows that there are a range of inter-warp load distances, which in many cases exceeds the reach of a next line or next two lines of prefetch. Table 1 shows the top four inter-warp load address distances found in each of the 14 evaluated GPU applications. Given that a L1 cache line is 128B, the next-line prefetching is effective when the distance is no longer than 2 × 128B. Therefore, the applications that have longer strides such as DCT and HSP cannot benefit from the next-line prefetcher.

Based on the above motivational data, in this paper we present CTA-aware prefetcher (CTAA) that addresses the limitation discussed. CTAA detects the base address of each CTA by concurrently running just one warp, called the leading warp, from each future CTA alongside the current CTA warps. The CTAA scheduler thus allows the base address to be computed earlier for the each CTA which then enables it to issue timely prefetches. Furthermore, to improve prefetch timing, we ad-
2. Prefetch background

2.1. Warp schedulers and prefetching

A warp is the smallest unit to execute instructions and GPU hides latency of operation with its simple SIMT pipeline structure by quick warp switching. A warp scheduler decides the priority of warps to select the warp to be issued among all the ready warps. A traditional round-robin scheduler fairly distributes the execution of warps by setting the priority of the most recently issued warp to the lowest. However, the round-robin scheduler isn’t effective to hide long latency of memory operation since it is highly probable that many warps executes the same load instructions within a short time window. It is because warps executes the same kernel and warps ordered in the round-robin fashion execute the same load instructions within a short time interval, as illustrated in Figure 5a, unless there is a large divergence in warps.

For the inter-warp stride prefetching, the detected stride among different warps is used to predict prefetching address by using a simple address prediction: \( \text{base address} + N \Delta \) where \( \Delta \) is the detected stride and \( N \) is the look ahead distance from the base where a prefetch is issued. For prefetching to be effective the base address must be computed first. For instance, in Figure 5a, if assume warp 0 (W0) effectively computes the base address immediately after executing \( L_{W0} \) and then computes the stride value \( \Delta \) after executing \( L_{W1} \) in warp 1, it can start issuing prefetches for W3 by using the simple address prediction, namely \( W0+3\Delta \). However the round robin scheduler schedules W3 fairly soon after and hence the corresponding demand request from the load \( L_{W3} \) will not gain much benefit from prefetching. Hence the prefetch request is ineffective as the prefetch distance (\( T_{D1} - T_{P1} \)) between the prefetch and demand request for the warp 3 is too short to hide long latency of the load instruction. Thus the distance between the prefetch and the demand request is too short under the traditional round-robin scheduler.

To hide long latency more effectively, the two-level warp scheduler is proposed [7, 16]. The two-level scheduler consists of two warp scheduling queues: a ready queue and a pending queue. In normal cases, warps in ready queue is selected to be issued with round-robin fashion. When a warp in the ready queue meets long latency load instructions with cache miss, the warp is demoted to the pending queue and one of warps in the pending queue is promoted. Thus, while the warp is waiting for data from the global memory, warps promoted to the ready queue executes many other instructions. Under this scheme, warps in different queues can execute different parts of a program, hence distance between load instructions from different warps gets longer. In Figure 5b, the stride is detected between the warp 0 and the warp 1 while they are in the ready queue and other warps are in the pending queue. The warp 0 and the warp 1 are demoted to pending queue after they issue a long latency load instructions. Simultaneously, prefetch requests for the warp 2 and the warp 3 are issued at \( T_{P2} \). Warp 2 and warp 3 are promoted to the ready queue which then execute their own computation phases while the prefetching requests are serviced by global memory. Therefore, the distance (\( T_{D2} - T_{P2} \)) between the prefetch and the demand requests can be longer under the two-level scheduler thereby improving prefetch effectiveness to hide long latency of load instructions. Thus warp scheduling plays an important role in improving prefetch timeliness and hence we use the improved two level scheduler as our baseline scheduler.

2.2. Effectiveness of prefetching

Prefetching doesn’t always provide positive effect on performance since inaccurate or unnecessary prefetches only increase memory traffic and cache data replacement [14, 22]. Following factors should be considered for positive impacts of prefetching.

**Accuracy:** Prefetch addresses are speculative since they are predicted based on previously tracked memory access patterns. For the stride or next-line prefetcher, accuracy of the predicted addresses is directly influenced by regularity of memory patterns. It is hard to detect address access regularity in GPUs as many threads accesses memory in parallel and thus temporality of access stream is highly perturbed in GPUs.
Coverage: Coverage of prefetching is directly associated with performance. Identifying base address and stride information is key to issue a prefetch for an upcoming demand fetch. As long as these two components are not detected prefetches cannot be issued.

Timeliness: If a prefetching request is issued too early than demand fetch, data from the prefetch requests may be evicted by other demand or prefetch requests before the corresponding demand fetch consumes the data. Also, a prefetch request cannot hide long latency of memory access effectively if the distance between prefetching and demand requests is short.

3. CTA-aware Prefetch

3.1. Overall operation

CTA-aware prefetcher (CTAA) exploits the regularity of memory patterns among warps 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. CTAA tracks the base address of each load by executing that load in a leading warp per CTA. In order to detect the base address and stride information as early as possible, the two-level warp scheduler is modified. Whereas the conventional two-level scheduler decides promotion and demotion of warps regardless of a CTA, the modified CTAA scheduler selects one warp from each CTA as a leading warp and schedules them for execution along with the ready warps from the current CTA by putting all these warps in the ready queue. Additionally, the modified CTAA scheduler increases the distance between prefetch and demand request to hide long latency of memory operation more effectively. Timeliness of prefetching is further enhanced by promoting a warp preferentially from pending queue to ready queue when prefetched data arrives in L1 cache. We will described each of these operational steps in detail.

3.2. Address prediction

As mentioned earlier, many GPU kernels use cooperative data loads executed across a massive number of threads rather than use loop iterations to load regular structured data. Figure 6 shows two example codes, from the LPS and BFS benchmarks, that do not use loop iteration for loading input data. The bold lines of the left hand side code box are the CUDA code statements that calculate the data indices. The right hand side shaded blue box shows the simplified index calculation of the same code. As shown in the code example, many GPU kernels use thread id and block id (or CTA id) to find the data index. Parameters such as blockId.X and blockId.Y are CTA-specific values that are constant across all warps within a CTA. On the other hand CTA parameters such as BLOCK_X and BLOCK_Y are compile-time known values that can be treated as fixed values across all CTAs within each kernel. Thus the component of a load address computation that relies on CTA-specific constant parameters can be represented as a constant values across all threads within a CTA. In the example, constant values computed from CTA-specific parameters are represented as \( C_1, C_2 \) and \( C_3 \). But each thread within a CTA has its own thread id and hence the effective index computation must use the thread id along with the constant value to compute the actual index. As all the CTAs of a kernel use the same number of threads and a fixed number of threads are grouped to form a warp, we conclude that the warps which have the same warp id in all the CTAs have the same load distance from the corresponding CTA’s base address which is calculated by using the CTA id and the input parameter values.

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 in each CTA is a fixed value, represented by the \( \Theta \) in the figure. 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 applied to all the CTAs. However, the CTA-specific constant values \( C_1, C_2 \) and \( C_3 \) must be computed for each CTA separately.

Note that the base address of CTA is more complex to predict even if it is a function of CTA id. For example, CTAs of (0,0), (3,3), (2,7) and (1,11) are all initiated in the same SM for LPS. But the \( \Delta \) value between consecutive CTAs is 5184, 6272 and 5824 respectively. Thus it is difficult to predict the base address of CTA from a prior CTA execution. It is observed that the offsets are not periodic and get more randomized as future applications generate more CTAs with highly diverging execution behavior.

Additionally, the \( \Delta \) values between CTAs can vary for different load instructions because different parameter values \( C_3 \) in the example of Figure 6) or various form of equation can be used to calculate base addresses. Based on these observation, the prefetch address of all the warps in a kernel can be calculated once one knows the load distance among the warps within a CTA and the base address of each CTA. The load distance can be dynamically computed by subtracting the load addresses of two consecutive warps (based on warp-ids) within the CTA for the load. CTAA detects CTA base addresses when at least one warp in a CTA executes a load instruction instead of using prediction or fixed computation logic.

We analyzed all 15 GPU applications we used in the study and we found that many of the load addresses used in the kernel have fixed distances. Only exception was the indirect references that graph applications normally use to find neighbor node and edges as shown in Figure 6b. \( g\_{\text{graph\_edges}}, g\_{\text{graph\_visited}}, g\_{\text{cost\[id\}}} \) and \( g\_{\text{updating\_graph\_mask}} \) are indexed by using the value loaded from \( g\_{\text{graph\_nodes\[tid\]}} \). Therefore, the address of these indirectly referenced variables are hard to pre-
dict. However, the metadata addresses (g_graph_mask[tid], g_graph_nodes[tid] and g_cost[tid]) can be calculated using thread id and CTA id as illustrated in the blue box and the fixed load distance is easily found.

### 3.3. Scheduling algorithm

Timing of prefetch is important for performance. The conventional two-level scheduler initially enqueues warps from each CTA to the ready queue in CTA orders; 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. This way, the leading warp of the trailing CTAs cannot be scheduled earlier than the trailing warps in the same CTA. To calculate the prefetch base address as early as possible, the leading warp of the trailing CTAs are enqueued right after the leading warp of the leading CTA. Figure 7 is the simplified illustration of the modified scheduling order. Each rectangle of the figure is a warp. Assume that a CTA consists of four warps and there are two CTAs. The left hand side figure presents that warps in the same CTA have fixed load distance (Δ) from the address of the leading warp of the CTA. To prefetch the three trailing warps’ load addresses of CTA 1, the leading warp of CTA 1 is scheduled right after the leading warp of CTA 0. Then, the ready queue size is filled such that the remaining warps of CTA 1 are scheduled later than the two leading warps from the two CTAs.

### 3.4. Dynamic leading warp nomination

The first warp of each CTA becomes the leading warp by default when a CTA is initiated in an SM. However, if there are diverged control flows, warp 0 may not cover all the diverged flows and hence may miss executing a load instruction that is necessary to compute the base address. To cover as many load instructions as possible, CTA dynamically nominates a leading warp for each CTA per load instruction. Whenever a warp encounters a load instruction for the first time in the CTA, the warp becomes a new leading warp. When a warp is newly nominated as a leading warp, the scheduler reorders the warps in the queues such that the new leading warp can be scheduled with the other leading warps back to back from that point of time. If the distance of the load instruction is already calculated by another CTA (which means the new leading warp is in a trailing CTA), only the new leading warp is reordered next to the other leading warps. Otherwise (which means the CTA of the new leading warp becomes a new leading CTA for that load instruction), the new leading warp as well as the trailing warps of the CTA are reordered such that the trailing warps are scheduled right after the leading warps to be used in load distance (Δ) calculation. The new warp order is maintained until a new leading warp is nominated for a future load instruction that is not executed in the current leader warp.

### 3.5. Hardware structure of the prefetcher

Figure 8 shows hardware structure of our CTA-aware prefetcher composed of two database tables (DIST and PerCTA), prefetch request generator and leading CTA/warp selector. The two-
level scheduler is also modified for the CTAA prefetcher. The DIST table in the CTAA prefetch logic logs the stride information computed after executing two warps from the same CTA. This table stores the PC and Δ which is the stride value for that load PC. When the same PC appears from two consecutive warps the Δ value is computed and stored in the DIST table. The DIST table has only two entries and thus at any time instance only two loads can be targets of a prefetch instruction. Although this approach may look overly conservative this approach works well for GPUs since many warps execute the same code within a short time window and hence the number of concurrent unique load PCs seen by the hardware is quite small.

<table>
<thead>
<tr>
<th>Table</th>
<th>Fields</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIST</td>
<td>PC (4B), stride (4B), miss counter (1B)</td>
<td>9B</td>
</tr>
<tr>
<td>PerCTA</td>
<td>PC (4B), leading warp id (1B), base address</td>
<td>139B</td>
</tr>
<tr>
<td></td>
<td>(32×4B), mask (6B)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Database entry size of prefetch engine

3.6. Detailed process of the prefetcher

Initiation of a leading warp: When a warp issues a load instruction the PC of the load instruction, memory address accessed the load instruction, CTA id, and warp id information are all delivered from the LD/ST unit to the CTAA prefetcher. The CTAA prefetcher then accesses the PerCTA table. Each CTA has its own PerCTA table and hence the CTA id is used to index the CTA specific table. Each table has two entries and each entry stores the load PC, leading warp id, and mask vector. The mask vector is 48 bits wide and each bit represents one warp id. Since each SM can have at most 48 warps, we use a 48-bit vector. But in practice the number of warps per CTA is less than 16 and hence even a 2B mask vector would suffice. The prefetcher uses issued load PC to do an associative search on the two entries of the PerCTA table. If no match is found then it assumes that this is the first time that load instruction is being executed in that CTA. It then randomly picks one entry from the PerCTA and uses that entry to store the newly issued load PC. It also stores the warp id and base address, which is the address accessed by the issued load. The mask bit associated with correspond warp id is also set to one.

Once a leading warp is entered into perCTA table the scheduler is informed about the leading warp and CTA ids so that it prioritizes executing the leading warp and at the same time it also reduces the priority of all other warps in the same CTA. Update of a leading warp: If a load PC matches one of the perCTA table entries then it is clear that some other warp has already reached this load instruction ahead of the current warp. In that case only the mask vector is updated to set the bit corresponding to the warp id that just executed the load instruction. The purpose of the mask vector is actually to detect loop instructions within a warp. Hence, when a load PC matches one of the perCTA table entries and the mask vector bit for that warp id is already set then it is clear that this same warp has already executed this load instruction once before. Hence, we assume that this load is being executed in a loop. But on each loop iteration the base address is different. Hence, when the mask vector bit for that warp id is already set then the base address is updated to indicate the new iteration of the loop. Furthermore, this warp is now treated as a new leading warp for that load instruction and the check mask is reset to all zeros except for the currently executing warp id. The renewed leading warp id information is again communicated to the scheduler.

Note that this scheme may not work well if different warps execute different loop iterations exclusively. For example, if there is a conditional load instruction in a loop, warp 0 and warp 1 may execute a load instruction in the first iteration and warp 2 and warp 3 may execute the same load instruction in the second iteration. However, when warp 2 issues the load instruction in the second iteration, the corresponding bit of the check mask will still be zero since it never executed the first iteration load. Hence, the base address is not updated in this case, even though it should have been updated for accurate prefetching. However, in practice we rarely noticed that these scenarios where loads in different loop iterations are exclusively executed by different warps. Hence, inaccurate prefetch addresses are rarely generated.

Issue of prefetch requests: Prefetches are triggered only when an actual load instruction from a warp is executed. There are two ways in which prefetches can be triggered. In the first case, a series of leading warps for multiple CTAs are in the ready queue followed by a series of trailing warps from just one CTA. When each leading warp for each CTA issues the load instruction it computes the base address and stores that information in the PerCTA table. Then the trailing warps for one CTA start executing the same load instruction. When a trailing warp from a CTA issues a load instruction the load PC is searched in all the PerCTA tables with the load PC. Each of the PerCTA table has a hit since the leading warp for that CTA has already issued the load and computed the base address. The same load PC is used to search the DIST table to obtain the Δ value. Thus the trailing warp of a leading CTA knows the base address of all the trailing warps of trailing CTAs. It then uses the base address and Δ value to compute the prefetch address and issues a prefetch request.

The second scenario for prefetching occurs when the leading warps are actually scheduled behind the trailing warps of the leading CTA. In spite of the best effort by the scheduler to prioritize the most important 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 CTA. In this case the trailing warp of leading CTA simply computes the Δ and leaves that information in the DST table. Later when a leading warp of a trailing CTA checks the PerCTA table it does not find any match. So it simply enters the base address
and load PC values into the PerCTA table, as explained before while describing the initiation of leading warp. In addition, it also checks the DST table with the PC and finds the \( \Delta \) value. In this case the leading warp of a trailing CTA issues multiple prefetch requests for all the warps within the trailing CTA. Note that some of these prefetches may be wasted since some of the warps in the trailing CTA may not execute the corresponding load instruction. However, prefetches are issued for all the warps within the trailing CTA in this second scenario.

The algorithmic flow of the above description is shown in Algorithm 1.

Algorithm 1 Control of CTAA prefetcher

```
if no matching PC in PerCta then
    register leading warp with current PC
    reset mask bits
else if valid matching PC in DIST then
    issue prefetches for current CTA
else if mask bit for warp id = 1 then
    update leading warp for the matched PC
    reset mask bits
else if valid matching PC in DIST then
    issue prefetches for current CTA
else
    if current CTA = leading CTA then
        calculate and update stride
        issue prefetches for valid base addresses
    else
        check prefetch and actual address
endif
set mask bit for warp id
```

Verification: To throttle inaccurate prefetches, the memory address of prefetch is verified by comparing with the address of the actual demand fetch. Thus the prefetch address is recalculated when an actual load instruction is issued. The counter value for miss-prediction is stored in DIST table for each load instruction. The miss counter increases by one whenever the calculated prefetch memory address is not equivalent to the demand fetch. When the counter value reaches a give threshold, the corresponding load instruction is excluded from the prefetch. The threshold value can be determined heuristically but a value of 3 is sufficient to throttle wasted prefetches. If threshold value is too high, inaccurate prefetches waste system resources and if it is too low, it may aggressively block a helpful prefetch.

3.7. Optimization

Warp wakeup on data arrival: To avoid prefetched data from evicted before consumption, the warps are waken 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 [10]. 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 alarmed to promote the warp that is bound to the prefetch memory request to the ready queue.

4. Evaluation

4.1. Settings and workloads

We implement the CTAA prefetcher on GPGPU-Sim v3.2.2 [4], a cycle-based GPGPU simulator. The baseline configuration is similar to Fermi architecture (GTX480) [19]. There are a total 15 SMs in the GPU device. Each SM has 32 SIMT lanes and a 16KB private L1 data cache. 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. Thus, the total size of L2 unified cache is 768KB. Timing parameters of the global memory is set based on GDDR5 with 924MHz memory clock [9]. Detailed configuration parameters is listed in Table 3.

We test 15 benchmarks selected from representative GPU benchmark suites as shown in Table 4. We collected the statistics by running the application either until the end of the execution or when the simulated instruction count reached 1B instructions. The change of performance by CTAA is compared to the baseline architecture using two-level warp scheduler with the ready warp queues having 8 entries. Additionally, several previously presented GPU prefetching methods are implemented to compare performance benefits of CTAA over prior work.

4.2. Performance enhancement

Figure 9 shows the reduction in execution cycles of prefetching methods normalized to the baseline configuration using two-level 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 in L1 data cache [11]. ORCH is the orchestrated prefetching where LAP is further enhanced with the prefetch-aware warp scheduling as described in [11]. SP means the simple next-line prefetcher which prefetches next cache line if one cache line is missed. 2xL1ID is the baseline configuration with two-level warp scheduler with twice the L1 cache size but with no prefetching.

Figure 9 shows execution time was reduced by up to 15.5% with CTAA and 5.4% on average over the baseline configuration. ORCH improves performance by about 1%. CTAA
### Table 3: GPGPU configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator</td>
<td>GPGPU-Sim v3.2.2</td>
</tr>
<tr>
<td>Core Configuration</td>
<td>1400MHz, 32 SIMT width</td>
</tr>
<tr>
<td>Resources per Core</td>
<td>48 concurrent warps, 128KB register file, 8 concurrent CTAs, 48KB shared memory</td>
</tr>
<tr>
<td>Scheduler</td>
<td>two-level scheduler (8 ready warps)</td>
</tr>
<tr>
<td>L1I cache</td>
<td>4-way set associative 4 sets of 128B blocks</td>
</tr>
<tr>
<td>L1D Cache</td>
<td>Private, 4-way set associative 32 sets of 128B blocks, LRU, 32 MSHR entries</td>
</tr>
<tr>
<td>L2D Cache</td>
<td>Unified, 8-way set associative 64 sets of 128B blocks per DRAM sub-channel, LRU, 32 MSHR entries</td>
</tr>
<tr>
<td>DRAM</td>
<td>924MHz, x4 interface, 6 channels, 2 sub-partitions per channel, FR-FCFS scheduler, 16 scheduler queue entries</td>
</tr>
<tr>
<td>GDDR5 Timing</td>
<td>( t_{CL} = 12, \ t_{RP} = 12, \ t_{RC} = 40, \ t_{RAS} = 28, \ t_{RCD} = 12, \ t_{RRD} = 6, \ t_{CDLR} = 5, \ t_{WR} = 12 )</td>
</tr>
</tbody>
</table>

### Table 4: Workloads

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmark</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA [1]</td>
<td>matrixMul</td>
<td>MM</td>
</tr>
<tr>
<td></td>
<td>dct8x8</td>
<td>DCT</td>
</tr>
<tr>
<td></td>
<td>scalarProd</td>
<td>SCP</td>
</tr>
<tr>
<td></td>
<td>fastWalshTransform</td>
<td>FWT</td>
</tr>
<tr>
<td>Mars [8]</td>
<td>InvertedIndex</td>
<td>IVI</td>
</tr>
<tr>
<td></td>
<td>Kmeans</td>
<td>KMN</td>
</tr>
<tr>
<td></td>
<td>PageViewCount</td>
<td>PVC</td>
</tr>
<tr>
<td></td>
<td>PageViewRank</td>
<td>PVR</td>
</tr>
<tr>
<td></td>
<td>SimilarityScore</td>
<td>SSC</td>
</tr>
<tr>
<td>GPGPU-Sim</td>
<td>LPS</td>
<td>LPS</td>
</tr>
<tr>
<td>bundle [4]</td>
<td>MUM</td>
<td>MUM</td>
</tr>
<tr>
<td>rodinia [5]</td>
<td>backprop</td>
<td>BP</td>
</tr>
<tr>
<td></td>
<td>heartwall</td>
<td>HW</td>
</tr>
<tr>
<td></td>
<td>hotspot</td>
<td>HSP</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>BFS</td>
</tr>
</tbody>
</table>

**Figure 9: Speedup of prefetcher over two level scheduler without prefetch**

**Figure 10: L1D and L2 cache miss reduction ratio**

**4.3. Increase of data traffic**

Figure 10 shows that L1 data cache and L2 cache miss ratio is decreased for most benchmarks with CTAA. On average, L1 data cache miss ratio is reduced by 4.8%. Surprisingly, miss ratio reduction of CTAA is similar to LAP and is actually worse than SP, which reduced the miss rate by 7.1%. Note that we count a cache hit whenever a tag matches 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 prefetch timeliness. Also, L2 cache miss ratio is reduced by 9.9% with CTAA and it is another positive factor for performance improvement.

Figure 11 shows data request packets from SMs to memory partitions.
partitions normalized to the baseline configuration. Prefetcher may generate extra data requests to waste resources of memory systems if inaccurate prefetch address is predicted or timing of prefetch request is not good. Increased data requests by CTAA is 0.5% that amount of data request packets with CTAA is nearly equivalent to the baseline configuration. It means CTAA doesn’t generate unnecessary or inaccurate prefetch requests.

4.4. Timeliness of prefetching

Right timing is critical for 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 cache accesses without benefit. The increased cache accesses may cause resource contention with the other memory requests thereby decreasing the performance. As stated in Section 3, 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 timing of prefetching and target load instructions effectively to improve performance. Figure 12a shows the percentage of early prefetches among the issued prefetches. The average of the early prefetch ratio is 1.71% and is less than 4% in most of the applications. There-fore, 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, prefetching requests should be issued sufficiently far ahead before demand requests are issued. Figure 12b shows the distance between prefetching and demand requests when CTAA is applied. On average, CTAA issues a prefetching request about 90 cycles before the targeted demand request.

4.5. Coverage and accuracy of prefetching

Figure 13 shows the coverage and accuracy of CTAA for 15 benchmarks. Coverage is defined as the ratio of prefetched memory requests among all data load requests. If a prefetch replaces a target memory request successfully, data is found in L1 data cache or the corresponding demand request is merged in the pending MSHR entry created by the earlier prefetching request (reservation hit). Therefore, coverage shows how many memory requests are substituted by prefetch requests. CTAA coverage ranges from 2% to up to 20%, with an average coverage of 10%.

Accuracy is the ratio of prefetching requests of which predicted addresses are identical to addresses of target load instructions. Accuracy is an important factor because unnecessarily prefetching data increases bandwidth and causes network congestion. As shown in Figure 13b, the accuracy of the CTA-aware prefetching is near 100% for most benchmarks. In the worst case the accuracy is around 70% for graph applications, such as IVI and KMN that have complex access patterns.

4.6. Impact on pipeline

RAW pipeline stall occurs when operands of an instruction are not ready when the instruction is issued to the pipeline. 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.
As shown in Figure 14, the RAW stall cycle is reduced by half in most of the applications when CTAA is used. The reduced stall cycle provides one reason for the observed performance improvements.

5. Related Work

Hardware prefetching is one of the prominent ways to overcome memory wall. Hardware and software prefetching approaches for data or instruction fetching have been studied and applied to modern microprocessors. We focus primarily on prior GPU prefetching and prior stride prefetching approaches since that is the focus of this work. If a data array has fixed stride and is accessed sequentially in a loop, a prefetching engine easily predicts addresses of next memory request based on base address and stride information. This simple prefetching, described in [21], can be easily implemented in CPUs. Baer and Chen [2] extended sequential prefetching to data arrays having variable strides. The address of prefetching request is predicted from previous address and strider information indexed by a PC of a load instruction. They distinguished states of prefetching since training time is required to get useful address and stride information. However, these prefetching schemes were developed in the context of single threaded applications. 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 [23]. Jouppi [12] proposed to add additional buffers to store prefetched data to prevent cache pollution. Srinath et al. [22] 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.

Several studies proposed memory prefetching algorithms for GPGPU [11, 15, 13, 24]. Woo and Lee [24] proposed a method to use idle GPU resource to accelerate CPU side memory performance. They exploited the large register file, hundreds of computing units, and high memory bandwidth in GPU. By assuming that CPU and GPU uses an unified memory space, GPU keeps track of cache misses and bring the data back to the L2 before future usage. By leveraging programmability of shader cores, various prefetching algorithms are implemented and dynamically triggered by OS whenever GPU has no pending work.

Lee et al. [15] proposed a software and hardware based many-thread aware prefetching which basically commands threads to prefetch data for the other threads so as to effectively reduce the cold misses. In software based prefetching, a prefetch instruction for next warp is added for each load instruction. By exploiting the fact that the memory addresses are referenced using thread id in many GPU applications, they simply predict the next warp’s memory request will be given to the data pointed by the next warp’s thread id. In the hardware prefetching, the basic purpose is the same that loads the next warp’s data but they employ a stride detector that keeps track of the load addresses so that the prefetch is issued only when there is a strided access pattern.

Jog et al. [11] pointed out that the current rotation based warp scheduling algorithms are not efficient to be used with conventional prefetching engines and 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 [13] 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. They pointed out that the GPGPU register file is highly underutilized. Instead of loading prefetched data to the smaller L1 cache which might cause cache thrashing, they store the prefetched data to the registers.

Most of the prior art devote efforts to augment the performance impact of simple prefectors by assuming that the pattern detection in GPGPU is complicated. Our approach opens a new direction of prefetching that predicts addresses of all the trailing warps by using base addresses of each CTA. Several optimizations and the modified scheduler improve the performance further.

6. Conclusion

This paper proposes a CTA-aware prefetch for GPUs. We show that prior stride based prefetching algorithms proposed in the context of CPUs cannot be easily applied in the context of GPUs. The thousands of concurrent threads that are present in GPUs and the complex scheduling algorithms obfuscate the stride behaviors in GPUs. Instead, we focus on the fact that all the CTAs assigned to a kernel execute the same code segment and the fact that the thread id and CTA id are commonly used in load address calculation in most of the kernel code. By leveraging this observation, CTA-aware prefetch predicts load
address of all the trailing warps by using the load distance detected by the leading CTA's execution and the base address of the trailing CTAs. By reordering the warps’ scheduling order, the prefetch distance is increased. The evaluation results show that the CTAA predicts prefetch addresses with over 94% accuracy and improves performance by 5.4% on average.

References