CTA-Aware Prefetching and Scheduling for GPU

Gunjae Koo*, Hyeran Jeon†, Zhenhong Liu‡, Nam Sung Kim‡, Murali Annavaram*
*University of Southern California, {gunjae.koo, annavara}@usc.edu
†San Jose State University, hyeran.jeon@sjsu.edu
‡University of Illinois at Urbana-Champaign, {zliu118, nskim}@illinois.edu

***Abstract*—Albeit GPUs are supposed to be tolerant to long latency of data fetch operation, we observe that L1 cache misses occur in a bursty manner for many memory-intensive applications. This in turn leads to severe contentions in GPU memory hierarchy, and thus stalls execution pipeline for many cycles as all warps end up waiting for their memory requests to be serviced by L1 cache. To spread such bursty L1 cache misses, we propose *CTA-Aware Prefetcher and Scheduler (CAPS)* consisting of a thread group-aware prefetcher and a prefetch-aware warp scheduler 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. CAPS allows prefetch requests to be issued sufficiently ahead of time before the demand requests, effectively reorganizing warp executions to quickly detect the base address of each CTA and stride per load. CAPS predicts addresses with over 97% accuracy and is able to improve GPU performance by 8% on average with up to 27% for a wide range of GPU applications.**

***Keywords*-prefetch; warp scheduling; GPU; SIMT**

I. Introduction

Long latency memory operation is one of the most critical performance hurdles in any computation. Graphics processing units (GPUs) rely on dozens of concurrent warps or wavefronts (a group of threads executed together) to hide the performance penalty of long latency operation by quickly context switching among available warps. When warps issue a long latency load instruction and following instructions are dependent on the load, 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 to minimize wasted cycles by long latency instructions. For instance two-level schedulers [1], [2] 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 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 critical bottleneck in GPUs, as has been identified in prior works [3], [4]. Especially, we observe GPUs cannot effectively hide long latency of load instructions as L1 cache misses occur in a burst manner for many memory-intensive applications. This leads to severe congestion in GPU memory subsystem to increase queuing delays *super-linearly*. Hence, many warps stall for hundreds of cycles as they end up waiting for its memory requests sent to L1 cache to be serviced. For instance our analysis for *nearest neighborhood*, a benchmark from CUDA SDK [5], reveals GPU pipelines are stalled for 62% of total execution cycles since all the warps end up waiting for the memory requests to be serviced from L1 cache. Such performance hurdle by memory operations will worsen as GPUs have supported more concurrent warps over generations while L1 cache size has not proportionally increased [6], [7]. Namely the number of L1 cache lines per warp has decreased, which leads to more bursty L1 cache misses due to severe interferences among warps [8].

To tackle this challenge researchers began to adopt memory prefetching techniques, which have been well-explored in the CPU domain, to the GPU domain. In CPU applications strided accesses are common across multiple iterations of a load in a loop. Accesses to data arrays found in iterative loops seen in CPU applications get spread over a large number of concurrent threads in GPUs. GPUs use the notion of thread blocks or cooperative thread arrays (CTAs) to manage these threads. Hence, inter-thread stride prefetching is a good way to capture the stride behavior in GPUs since regular strides are detected among different threads which access these data arrays indexed as a function of thread id. Prefetches can be issued for trailing threads using the base addresses and strides computed from the currently running threads [9]. Since GPUs execute a warp (or wavefront) as the basic execution unit inter-thread prefetching essentially must be implemented as inter-warp prefetching.

One of the fundamental challenges with prefetching is the tradeoff between accuracy and prefetch distance. Accuracy

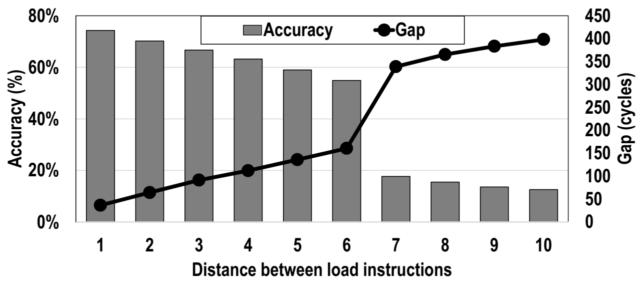
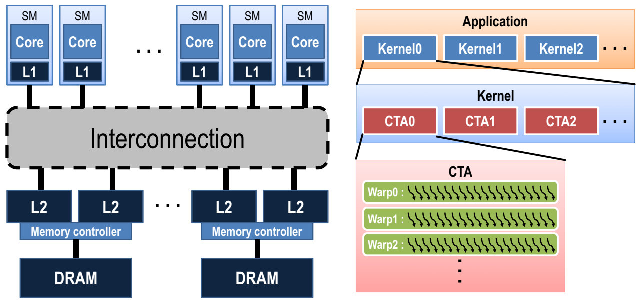


Figure 1: Accuracy with stride-based inter-warp prefetch and cycle gaps by distances of warps

is measured as the fraction of demand fetches that were correctly targeted by prefetching. Prefetch distance is defined as the cycle gap from when the prefetch is issued to when the demand fetch requests the data. Figure 1 shows the accuracy and prefetch distance of the simple inter-warp stride prefetching when warps execute loads of the same PC for the stride-friendly benchmark *matrixMul*. The x-axis in the figure indicates differences of warp ids from the baseline warp to the target warp. The primary y-axis shows the accuracy. When the distances between warps are short then accuracy is high. The line plot shows the prefetch distance, which measures the difference of clock cycles between load execution by the baseline warp and the target warp. If the distance is just one, prefetch can be performed targeting the very next warp, then the number of cycles between the prefetch and demand fetch is just a few tens of cycles. Since global memory access takes hundreds of cycles a short prefetch distance cannot hide this access latency. In order to increase the distance between prefetch and demand fetch one has to increase the distance between the baseline and the target warps. But as we move along the x-axis prefetch accuracy drops gradually and then suffers a steep drop at a distance of seven.

Further analysis shows that the main culprit for this drastic reduction in accuracy is that at the beginning of each CTA boundary the base address of the stride changes. When a prefetch is issued for a target warp in a different CTA, the prefetch address does not match the demand fetch. Since *matrixMul* has 8 warps per CTA, over the distance of seven every prefetch crosses the CTA boundary and the accuracy drops dramatically. That means the inter-warp prefetching is accurate for just a few neighboring warps within each CTA, which have low potential to hide memory latency, or the technique may send inaccurate prefetch requests, which will negatively impact performance.

In order to overcome the limitations in accuracy and timeliness of prefetching in GPU, we present *CTA-Aware Prefetcher and Scheduler (CAPS)*. The prefetch engine computes accurate prefetch addresses by detecting a baseline address per CTA and a common stride in data arrays. The prefetch-aware warp scheduler works in conjunction with the prefetcher to improve the timeliness of prefetching.



(a) GPU architecture (b) GPU application structure

Figure 2: GPU hardware and software architecture

II. Background

A. GPU Execution Model

Figure 2a shows the GPU hardware architecture composed of multiple streaming multiprocessors (SMs) and memory partitions [10], [6]. Each SM has 32 single instruction multiple thread (SIMT) lanes where each SIMT lane has its own execution units, labeled as cores in the figure. Each SM is associated with its own private memory subsystem, a register file and L1 caches. SMs are connected to L2 cache partitions shareable across all SMs via an interconnection network. L2 cache banks connect to global memory through one or more memory controllers. If data is fetched from the global memory, the latency, which varies with data traffic, is hundreds or even thousands of core cycles [11].

The GPU software execution model is shown in Figure 2b. A GPU application consists of several kernels, which are basic task modules that exhibit a significant amount of parallelism. Each kernel is split into groups of threads called thread blocks or CTA. A CTA is a basic workload unit assigned to an SM. Threads in a CTA are sub-grouped into a warp, the smallest execution unit sharing the same PC. 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 one cache line request. Therefore, only one or two memory requests can be generated if requests in a warp are regularly distributed.

B. 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. The GPU hardware 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

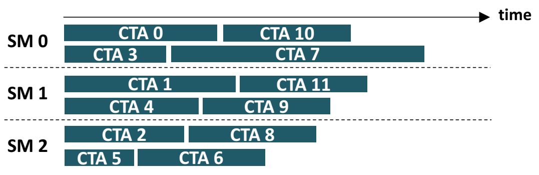


Figure 3: Example CTA distribution across SMs

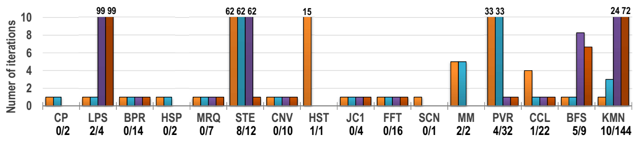


Figure 4: The average number of iterations for load instructions in a kernel. Repeated load instructions / total load instructions (by PC) under names of benchmarks

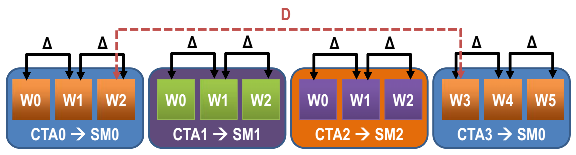


Figure 5: Irregular stride among CTAs assigned to the same SM

are assigned up to the maximum concurrent CTAs that can be accommodated in an SM. Once each SM is assigned the maximum allowed CTAs a new CTA is assigned to an SM only when an existing CTA on that SM finishes execution. As a result, irrespective of how the initial CTA assignment process starts eventually CTA assignments to SMs are purely demand-driven.

Figure 3 shows an example CTA distribution across three SMs. Assume that a kernel consists of 12 CTAs and each SM can run two concurrent CTAs. At the beginning of the kernel execution, SM 0, 1, and 2 are assigned two CTAs, one at a time in a round-robin fashion. 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, and then CTA 7 is assigned to SM 0 after termination of CTA 3. Therefore, CTA assignments to an SM are determined dynamically based on CTA termination order.

III. Limitations of Prefetches in GPU

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

A. *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 [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 applications which have iterative irregular and diverged memory accesses [13].

The effectiveness of the intra-warp prefetching depends on the presence of load instructions that are repeatedly executed in loops. But there is a growing trend towards replacing deep loop operations in 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 4 shows the average iteration number of the four common loads in the selected benchmarks, which are described in Section VI. We measured the execution number of each load, distinguished by the PC value, in a warp and picked the four most frequently executed loads. If a load instruction is part of a loop body then that PC would have repeatedly appeared in the execution window. Also the number of loads within loops over the total loads found in a kernel code is also shown under each benchmark name on the x-axis. These results show that when a loop intensive CPU program is ported to CUDA (or OpenCL), loops are reduced to leverage massive thread level parallelism. This observation has also been made in a prior study that showed deep loop operations are seldom found in GPU applications [14], [15].

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 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.

B. *Inter-Warp Stride Prefetching*

Strides existing among warps can be extended to inter-warp stride prefetcher approaches [9], [16]. If regular offsets of memory addresses are detected between warps, then inter-warp prefetching detects a base address and a stride value across different warps based on warp id. Thus inter-warp stride prefetcher issues prefetches for future warps from a current warp using the base address and warp-id differences. The inter-warp prefetcher has potential to cover thousands of threads if prefetch requests are predicted correctly as a number of concurrent warps supported by GPU hardware increases. (Fermi: 48, Kepler: 64) [10], [6].

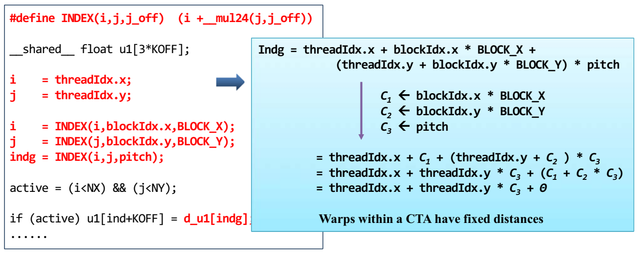
The CTA distribution algorithms employed in current GPUs limits applicability of inter-warp stride prefetching to warps within a CTA. As shown in Figure 3, SMs are not assigned consecutive CTAs. Thus within a CTA all the warps are able to see stride accesses but the prefetcher is unable to prefetch across CTAs assigned to the same SM. This is because inter-warp prefetchers simply expect continuous warps access the next stride, regardless what CTA each warp belongs to. Even if we assume that there is an inter-warp stride prefetcher that also considers CTA id, as each SM does not run consecutive CTAs, it is hard to predict the accurate address without knowing the inter-CTA stride. The inter-CTA stride may differ from inter-warp stride, which means that the base address of a CTA is not always distant by inter-warp stride from the address of the last warp of the previous CTA, because load addresses are typically calculated by using CTA id aside from thread id and warp id. Thus accurate prefetch is limited to warps within a CTA. As quantified earlier in Figure 1 inter-warp prefetching suffers from loss of accuracy as we go across CTA boundaries.

C. 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 policies for GPUs was proposed in [17], [18]. 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 next-line prefetching scheme, prefetch requests are issued too close to the demand fetch, resulting in small performance improvements.

IV. Where Did My Strides Go?

In this section we provide some insights into how GPU execution model perturbs stride access patterns seen at the application level. Figure 6 shows two example codes, from the LPS and BFS benchmarks [19], [20]. 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 box represents the corresponding equation to show how the array indexes will be computed. 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 [21]. Parameters such as BLOCK_X and BLOCK_Y are compile-time known values



```
#define INDEX(i,j,j_off)  (i +__mul24(j,j_off))

__shared__ float u1[3*KOFF];

i    = threadIdx.x;
j    = threadIdx.y;

i    = INDEX(i,blockIdx.x,BLOCK_X);
j    = INDEX(j,blockIdx.y,BLOCK_Y);
indg = INDEX(i,j,pitch);

active = (i<NX) && (j<NY);

if (active) u1[ind+KOFF] = d_u1[indg];
......
```

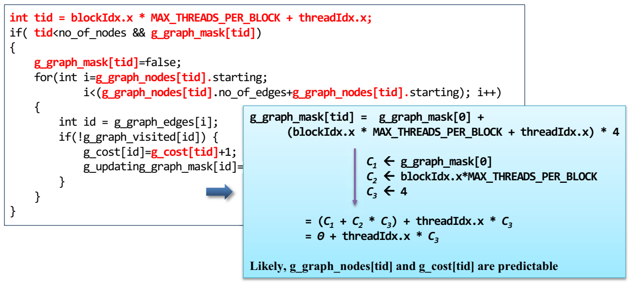
```
Indg = threadIdx.x + blockIdx.x * BLOCK_X +
         (threadIdx.y + blockIdx.y * BLOCK_Y) * pitch

            C1 ← blockIdx.x * BLOCK_X
            C2 ← blockIdx.y * BLOCK_Y
            C3 ← pitch

       = threadIdx.x + C1 + (threadIdx.y + C2 ) * C3
       = threadIdx.x + threadIdx.y * C3 + (C1 + C2 * C3)
       = threadIdx.x + threadIdx.y * C3 + θ
```

Warps within a CTA have fixed distances

(a) LPS [19]



```
int tid = blockIdx.x * MAX_THREADS_PER_BLOCK + threadIdx.x;
if( tid<no_of_nodes && g_graph_mask[tid])
{
    g_graph_mask[tid]=false;
    for(int i=g_graph_nodes[tid].starting;
            i<(g_graph_nodes[tid].no_of_edges+g_graph_nodes[tid].starting); i++)
    {
        int id = g_graph_edges[i];
        if(!g_graph_visited[id]) {
            g_cost[id]=g_cost[tid]+1;
            g_updating_graph_mask[id]=
        }
    }
}
```

```
g_graph_mask[tid] =  g_graph_mask[0] +
      (blockIdx.x * MAX_THREADS_PER_BLOCK + threadIdx.x) * 4

            C1 ← g_graph_mask[0]
            C2 ← blockIdx.x*MAX_THREADS_PER_BLOCK
            C3 ← 4

       = (C1 + C2 * C3) + threadIdx.x * C3
       = θ + threadIdx.x * C3
```

Likely, g_graph_nodes[tid] and g_cost[tid] are predictable

(b) BFS [20]

Figure 6: Load address calculation examples

that can be treated as fixed values across all the CTAs in each kernel. Parameters such as blockId.X and blockId.Y are CTA-specific values that are constant only across all the threads in a CTA. Thus load address computations rely on a mix of constant parameters, CTA-specific parameters 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 the constant parameter used 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 represented as $\theta = C_1 + C_2 \times C_3$. Once a CTA's base address is computed, each thread can then use its thread id (represented by threadIdx.x and threadIdx.y) and the stride value represented by C_3 to compute the effective array index.

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 of which distance between consecutive warps is one. Therefore, the load address difference 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 a CTA is cannot be predicted easily even if it appears to be a function of CTA id. Because

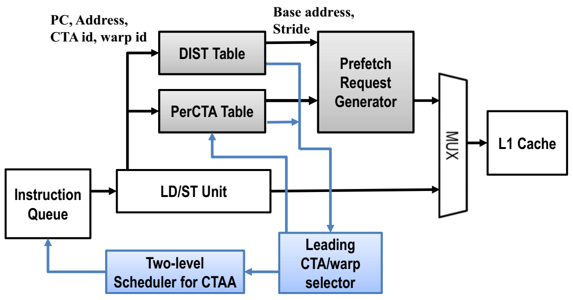
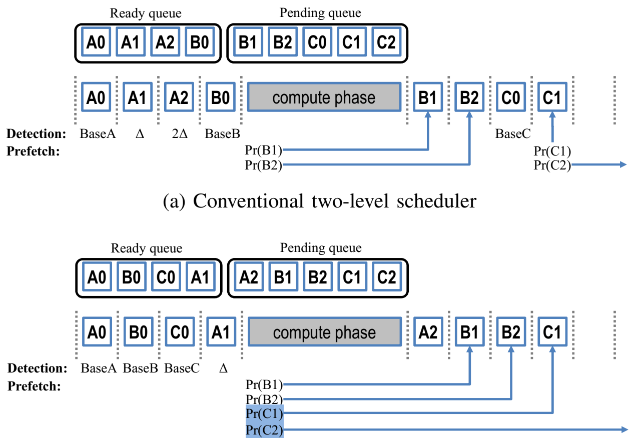

PC, Address, CTA id, warp id
Base address, Stride
DIST Table
PerCTA Table
Prefetch Request Generator
MUX
L1 Cache
Instruction Queue
LD/ST Unit
Leading CTA/warp selector
Two-level Scheduler for CTAA

Figure 7: Hardware structure of CAPS



(a) Conventional two-level scheduler

(b) Prefetch-aware two-level scheduler

Figure 8: Prefetch-aware warp scheduler

this function varies from one load to another load instruction in the same kernel, and differs across kernels. Also inter-CTA distances (difference of base addresses between two CTAs) in an SM is irregular. For example, CTAs (0,0), (3,3) and (7,2) are all initiated in the same SM for LPS in our simulation run. 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 the first warp of 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 within the CTA for the same 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 a CTA. One exception is the indirect references that graph analytics applications normally use to find neighboring node and edges as shown in Figure 6b. In the BFS code $g\_graph\_visited$ is indexed by variable id which is a value loaded from $g\_graph\_edges[i]$. Therefore, the address of these indirectly referenced variables cannot be predicted using stride prefetcher. However, the metadata addresses (g_graph_mask, g_graph_nodes and g_cost) are all thread-specific references and these addresses can be calculated using thread id and CTA id.

V. CTA-Aware Prefetcher and Scheduler

In order to increase the accuracy of prefetching, *CTA-Aware Prefetcher (CAP)* computes prefetch addresses across all concurrent warps in an SM by detecting address changes across CTA boundaries, and regular strides between warps within a CTA. In addition, *Prefetch-Aware Scheduler (PAS)*, which can be implemented as a simple enhancement to the two-level scheduler, cooperates with CAP to improve the timeliness of prefetching. Figure 7 shows overall hardware structure of CAPS. The prefetcher relies on PerCTA table to store the base address of each CTA which is computed as early as possible using a leading warp from each CTA. The DIST table is a single global structure across all CTAs and it tracks the stride distance for a few loads that are selected for prefetching. The prefetch request generator consists of simple adder logic blocks to compute prefetch addresses using the base addresses and the strides from PerCTA and DIST tables. Prefetch requests access L1 data cache with lower priority than demand fetches. We will now describe the operation of the two components in details.

A. Prefetch-Aware Scheduler

CAP requires the base address of each CTA and the stride between two warps within a CTA to generate prefetch requests for all warps across other CTAs. CAP tracks the base address for a given CTA by executing one warp. We call the warp that computes the base address of a given CTA as the leading warp, W_{lead}. When a load of a certain PC is issued for the first time, the CTA containing the warp issuing the load is set as the leading CTA, say CTA_{lead}. The warp issuing the load earliest becomes the leading warp of CTA_{lead} naturally. Once we execute one more warp from CTA_{lead} then it is possible to compute the stride value. Once the stride value is computed then we need to compute the base address of each trailing CTA as early as possible. In this paper all other CTAs except the CTA_{lead} will be referred to as the trailing CTAs, CTA_{trail}. We pick one warp from each of the trailing CTAs to compute the base addresses. Each warp that is picked to compute the base address of the corresponding trailing CTA will become the leading warp of that trailing CTA. Using this terminology we describe the operation of PAS.

Let's assume that 3 CTAs (CTA **A**, CTA **B**, and CTA **C**) are running concurrently in an SM and each CTA is composed of 3 warps that exhibit stride behavior. The conventional two-level scheduler initially enqueues warps from each CTA to the ready queue in CTA order; warps of the CTA **A** are first enqueued to the ready queue and then the warps of the following CTAs are enqueued until the ready queue is filled up as shown in Figure 8a. In the example scenario warp $A0$ issues the load for the first time thus the base address of the CTA **A** is detected. The stride is also detected when warp $A1$ executes. Even though we have the stride value the base address of CTA **A** is not useful for computing the prefetch address for CTA **B** and CTA **C**. Hence only when warp $B0$ is executed the base address of CTA **B** is known. Then it is possible to issue prefetches for the other warps ($B1$ and $B2$) in CTA **B**. Similarly only when $C0$ executes the base address of CTA **C** is known and prefetches can be issued for $C1$ and $C2$, but the prefetch distance is quite small since these warps are executed back-to-back. If the base addresses of all CTAs are computed eagerly by the two-level scheduler then prefetches could have been issued much earlier.

Figure 8b shows the modified prefetch-aware scheduler. One leading warp ($A0$, $B0$ and $C0$) is selected from every CTA and these are enqueued in the ready queue first. Then warp $A1$ fills the remaining slots of the queue. As $A0$, $B0$ and $C0$ issue load instructions, base addresses of CTA **A**, CTA **B** and CTA **C** are computed eagerly. The stride can be computed after the issuance of $A1$. Since we know the base addresses of all CTAs and the stride value is same across all CTAs, prefetch addresses can be computed for other warps ($B1$, $B2$, $C1$ and $C2$) of trailing CTAs.

Scheduler implementation: PAS is a simple enhancement to the conventional two-level scheduler. To implement the desired functionality we divide the ready queue into leading warp queue and trailing warp queue. The pending queue design is unaltered. One warp from every CTA is marked as a leading warp using a one-bit leading warp marker. PAS pushes warps that have this marker set to the front of the ready queue. Unlike the conventional two-level scheduler that chooses the oldest ready warp, PAS chooses the leading warps first. Note that while we discussed PAS implementation on top of a two-level scheduler it is also possible to make simple enhancements to the loose round-robin scheduler to achieve the same effect of prioritizing leading warps. Also, in the GTO, when a warp is greedily scheduled until it encounters a memory operation, our approach can be applied by prioritizing the leading warps so that the leading warps are greedily scheduled until they compute the base address. Then the trailing warps can continue to execute. In our evaluations we implement the algorithm on the top of two-level scheduler to test the impact of PAS for better timeliness because the two-level scheduler already mitigates hurdles of memory operations by dispersing data fetch groups [2], [17].

Warp wake-up: To avoid eviction of prefetched data before demanding, 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 one of ready warps forcibly into the pending queue. The similar approach was proposed by OWL [18]. Only minimal change is needed for implementing the eager warp wake-up. 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 wake-up, 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.

B. CTA-Aware Prefetcher

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 from the 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 four distinct loads (identified by their program counters) within each CTA. Hence, the PerCTA table has four 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 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 CTA. 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 limited number of 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 up to four different values across the maximum of four distinct memory requests from the same load instruction. If the stride values 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.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.

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. 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 one byte and the threshold is set to 128 by default.

The entry structure of PerCTA and DIST tables described in this section is summarized in Table I.

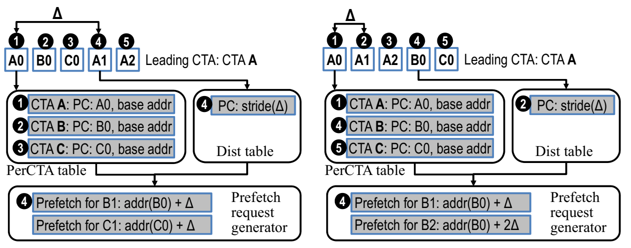
Table	Fields	Total
PerCTA	PC (4B), leading warp id (1B), base address (4×4B)	21B
DIST	PC (4B), stride (4B), mispredict counter (1B)	9B

Table I: Database entry size of the prefetcher

Handling indirect accesses: In indirect accesses regular strides between different warps are rarely observed since addresses of the indirect accesses are computed from random data fetched by other global loads as shown in the example of BFS code (Figure 6b). Thus incorrect prefetch requests for the indirect accesses can degrade the performance of memory systems, furthermore prefetcher resource is wasted. In order to avoid such situation indirect accesses are detected and excluded from prefetch process. Indirect accesses can be detected by tracing the source registers of global loads backward [22]. If the source register of a load is originated from thread ids, block ids, and constant parameters, CTA-aware prefetcher can predict prefetch addresses. Otherwise, the load is not considered for prefetch.

C. 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


Δ
1 2 3 4 5
A0 B0 C0 A1 A2 Leading CTA: CTA A
1 CTA A: PC: A0, base addr
2 CTA B: PC: B0, base addr
3 CTA C: PC: C0, base addr
PerCTA table
4 PC: stride(Δ)
Dist table
4 Prefetch for B1: addr(B0) + Δ
Prefetch for C1: addr(C0) + Δ
Prefetch request generator
Δ
1 2 3 4 5
A0 A1 A2 B0 C0 Leading CTA: CTA A
1 CTA A: PC: A0, base addr
4 CTA B: PC: B0, base addr
5 CTA C: PC: C0, base addr
PerCTA table
2 PC: stride(Δ)
Dist table
4 Prefetch for B1: addr(B0) + Δ
Prefetch for B2: addr(B0) + 2Δ
Prefetch request generator

(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

order of each warp's load instruction issuance. We assume warps $A0$, $B0$ and $C0$ have already finished execution and they have updated the PerCTA table. But there is no stride value that is stored in the DIST table as yet since none of the trailing warps has been executed. When $A1$, which is a trailing warp of CTA **A**, issues the load instruction the stride (Δ) value is computed. Then the prefetcher traverses each of the PerCTA tables with the PC value of $A1$. 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 CTA **A** and CTA **B**, prefetch requests for $B1$ and $C1$ 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 executed behind the trailing warps of the leading CTA. In spite of the best effort by PAS 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 $A1$ executes ahead of $B0$ and $C0$. As $A1$ of CTA **A** issues a load instruction before $B0$ and $C0$ of CTA **B** and CTA **C**, then Δ is computed and stored in DIST table before the PerCTA table is updated with base addresses for CTA **B** and CTA **C**. After $B0$ is issued with updating PerCTA table with the base address as in the first scenario, prefetch requests for other warps in CTA **B** are also generated by using the stride value that is already computed in DIST table. Thus in this scenario the leading warp of a trailing CTA enables to issue prefetches for all the trailing warps of its own CTA.

D. Hardware Cost

CAPS 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 four entries per PerCTA and four 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 708 Bytes as summarized in Table II.

Table	Configuration	Total
DIST	9 bytes per entry, 4 entries	36 bytes
PerCTA	21 bytes per entry, 4 entries, 8 CTAs	672 bytes

Table II: Required hardware for tables

We modeled CAPS in RTL level and synthesized with FreePDK 45nm library [23] to estimate hardware cost of CAPS. We assumed DIST table consists of arrays of simple flip-flops since the required data space for DIST is small. We used CACTI [24] to estimate area and power dissipation of perCTA table, which needs more data entries because an SM runs multiple CTAs concurrently. The estimated area of CAPS is 0.018 mm^2, which occupies 0.08% of one SM given that the area of one SM measures 22 mm^2 base on the die photo of GF100. Under the 45 nm FreePDK library and CACTI configuration it is estimated that CAPS consumes 15.07 pJ per access and 550 μW of static power.

VI. Evaluation

A. Settings and Workloads

We implemented CAPS on *GPGPU-Sim v3.2.2* [19]. The baseline configuration listed in Table III is similar to Fermi (GTX480) [10]. Detailed configuration parameters are listed in Table III. We chose Fermi architecture because GPGPU-Sim generates performance statistics with very high correlation with actual NVIDIA Fermi GPU [19]. However, our prefetching mechanism does not depend on a specific architecture because CAPS exploits base address computations for concurrent CTAs, which don't change in newer GPU architectures. Likely, we believe our scheme is applicable to other architectures that run a kernel in multiple groups. For example, AMD GPUs execute OpenCL programs run by multiple workgroups, which are equivalent to CTAs, and a workgroup is split into wavefronts, basic groups of threads scheduled in compute units [25]. Even though AMD APU uses a unified system memory that can be accessed by both CPU and GPU, the GPU side memory access patterns are determined by the application characteristics.

Parameter	Value
Core	1400MHz, 32 SIMT width, 15 cores
Resources / core	48 concurrent warps, 8 concurrent 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, $\times$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

Table III: GPU configuration

We used 16 benchmarks selected from various GPU benchmark suites as listed in Table IV. Most of the benchmarks are memory latency sensitive applications whose performance is improved with reduced memory channel delays, thus the influence of reduced memory latency can be tested. We also studied irregular applications (PVR, CCL, BFS, and KM) to test the quality control mechanism of CAPS for divergent memory accesses [26]. All applications were simulated until the end of their execution or when the simulated instruction count reached one billion. CAPS 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 CAPS.

Benchmark	Abbr.	Benchmark	Abbr.
Coulombic Potential [19]	CP	laplace3D [19]	LPS
backprop [20]	BPR	hotspot [20]	HSP
mri-q [27]	MRQ	stencil [27]	STE
convolutionSeparable [5]	CNV	histogram [5]	HST
jacobi1D [28]	JC1	FFT [29]	FFT
scan [5]	SCN	MatrixMul [5]	MM
PageViewRank [30]	PVR	Connected Comp. Label [31]	CCL
Breadth First Search [20]	BFS	Kmeans [30]	KM

Table IV: Workloads

B. Performance Enhancement

Figure 10 shows normalized IPC of various prefetching methods to the baseline configuration using two-level warp scheduler without prefetching. CAPS is our proposed CTA-aware prefetcher and scheduler. INTRA is a simple intra-warp stride prefetching introduced in Section III-A, which produces prefetch requests when iterative loads have regular strides in a loop. INTER is an inter-warp stride prefetching to generate prefetch requests for loads having regular strides across warps as described in Section III-B. MTA is the many-thread aware prefetching combining both intra-warp and inter-warp stride prefetching as described in [9], which applies the intra-warp prefetching for iterative loads having regular strides within a warp, otherwise makes prefetch requests for other warps when regular address distances are detected among warps. Specifically, we implemented a hardware-based prefetcher among various mechanisms presented in [9]. NLP means the simple next-line prefetcher introduced in Section III-C, which sends prefetch requests for the next cache line if one cache line is missed. LAP is the locality-aware prefetching built on top of the 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].

Figure 10 shows CAPS improves overall performance by 8% on average, with up to 27% performance improvement for CNV. Performance of irregular applications is improved

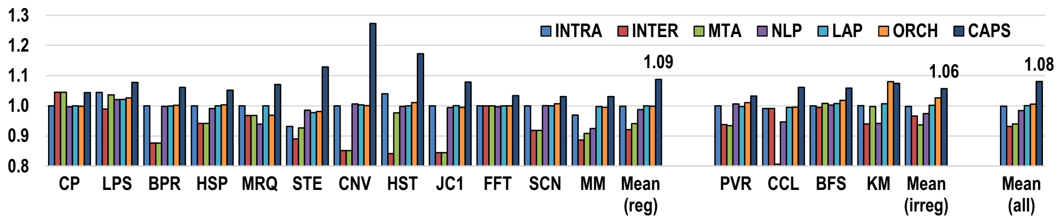


Figure 10: Normalized IPC over two level scheduler without prefetch

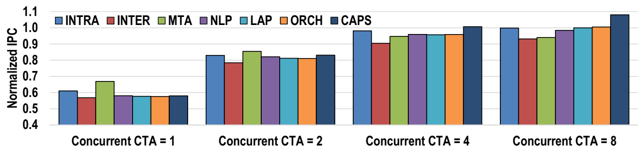
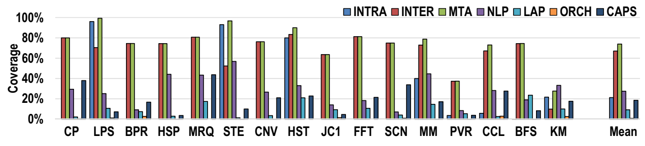
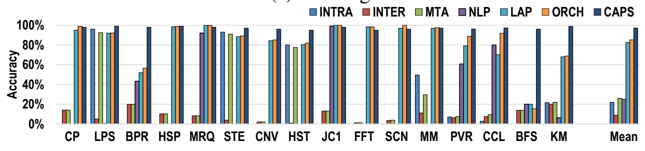


Figure 11: Performance by number of concurrent CTAs



(a) Coverage



(b) Accuracy

Figure 12: Prefetch coverage and accuracy

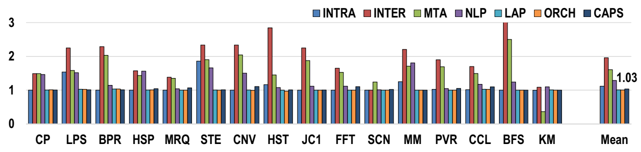
by 6% with CAPS, which represents irregular applications can be benefited by accurate prefetch for strided loads while throttling divergent memory requests from indirect accesses. INTRA shows performance improvement for several applications since INTRA uses intra-warp prefetching which tend to benefit only loop intensive kernels; and in the absence of loops intra-warp prefetching by itself is ineffective. Performance benefit by INTER is negative as it produces incorrect prefetch requests for trailing warps over CTA boundaries as described in Section III-B. The performance gain by MTA is not better than INTRA since MTA uses inter-warp prefetching if iterative loops are not detected within a warp and inter-warp prefetching causes negative effect with multiple concurrent CTAs running in an SM. NLP doesn't provide better performance either because prefetch requests for next lines in the case of cache misses guarantee neither accuracy or timeliness. LAP and ORCH improve performance by about 1% and it is lower than the performance improvements reported in [17]. Note that in [17] the larger performance improvements were shown on top a round robin scheduler. The authors also showed that when a 2-level warp scheduler is used in the baseline the performance improvements are much smaller. Hence, the 1% performance improvement is in line with prior work results.

Figure 11 shows the performance of prefetchers by the number of CTAs assigned to each SM. By default Fermi allows 8 CTAs, and newer Kepler architecture runs up to 16 CTAs per SM [10], [6]. The increasing CTA count accommodated per SM only makes the CTA-aware prefetching even more critical. The figure shows the average IPCs normalized to the average IPC of the base configuration (the maximum concurrent CTAs = 8). As shown in Figure 11 performance of INTRA and MTA is better than other prefetchers when only one CTA is allowed. The reason for this higher performance is that these prefetchers do not have to cross CTA boundaries. CAPS does not provide any benefits since by definition it prefetches across CTAs. But note that curtailing CTAs to remove uncertainty in data accesses at the CTA boundaries is not beneficial. The performance of all prefetchers is worse than the baseline that uses 8 CTAs without any prefetching. Hence, increasing CTA count improves performance that cannot be overlooked. As the number of concurrent CTAs increases CAPS outperforms other prefetchers. On the other hand, the performance of MTA degrades because of increasing discontinuities at CTA boundaries. This performance trend shows why CAPS works well for modern GPU architecture which exploits massive thread parallelism using many concurrent CTAs.

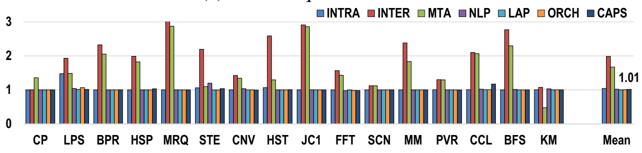
C. *Coverage and Accuracy of Prefetching*

Figure 12 shows the coverage and accuracy of prefetchers. Coverage is defined as the ratio of the number of issued prefetch requests compared to the total demand fetch requests. Higher coverage ratio doesn't always improve performance since inaccurate prefetches only consume resources like cache space and memory channel bandwidth. Accuracy is the ratio of correctly estimated prefetch requests that were actually consumed by the demand requests. Accuracy is an important performance factor of a prefetcher because unnecessarily prefetched data increases bandwidth and cause cache pollution. GPU's memory subsystem is remarkably vulnerable to inaccurate prefetches since local cache resources are already under immense pressured due to requests from thousands of threads in an SM.

CAPS on average provides 18% coverage. Coverage is lower for PVR, BFS, and KM that traverse graphs and have irregular data access patterns caused by indirect memory accesses. These loads are excluded from prefetch, thereby curtailing coverage. Nevertheless, CAPS can test and generate prefetch requests with high accuracy ratio for



(a) Fetch request from cores



(b) Data read from memory

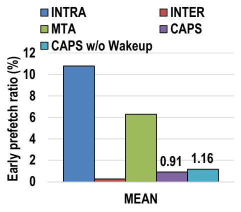
Figure 13: Bandwidth overhead by prefetching

strided loads in irregular applications. CAPS has also low coverage ratio for HSP. These applications have irregular strides between warps, thus CAPS recognizes mismatches between prefetch and demand requests and then avoids issuing wasteful prefetch requests quickly.

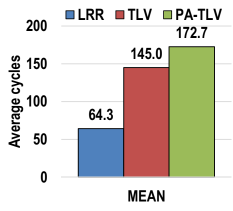
The accuracy of CAPS is very high for most benchmarks. Even though divergent memory accesses are frequently observed in irregular applications, CAPS employs the throttling mechanism to avoid indirect accesses which will result in inaccurate prefetch requests. Furthermore CAPS uses the PC-based detection mechanism for inaccurate prefetch requests, described in Section V-B, to recognize mispredicted prefetch requests quickly and shuts down prefetching for the corresponding loads. Thus CAPS has 18% of coverage with very high accuracy (97%). Low accuracy is the main reason of the low performance of INTER and MTA despite their high coverage ratio. INTRA has decent accuracy and coverage if applications have deep iterative loops and regular access patterns like LPS and STE, nevertheless, the performance of INTRA is constrained by a large amount of useless prefetch resulted from too early prefetch as shown in Figure 14a.

D. Bandwidth Overhead

Data bandwidth increase is prefetcher's prominent overhead since additional request and data traffic consumes resources in memory channels. Figure 13 presents the bandwidth overhead of prefetchers. Figure 13a and Figure 13b respectively show increased data request traffic from SMs to memory partitions, and increased read data traffic from DRAM compared to a baseline with no prefetching. If prefetch requests are actually consumed by a later demand fetch, the increase in data traffic should be minimal. Otherwise, wasted prefetches will increase data bandwidth which degrades performance. As shown earlier, CAPS has very high accuracy. Hence, most of the prefetched data is consumed by a demand fetch later. INTER on the other hand increases traffic by over 2× since its coverage is high but accuracy is low. MTA also increases data bandwidth significantly for the same reason. The bandwidth overhead of CAPS is less than 3%.



(a) Early prefetch ratio



(b) Prefetch distance of timely prefetches

Figure 14: Timeliness of prefetching

E. Timeliness of Prefetching

When a prefetch is issued too early, the prefetched data can be evicted before the target demand load is issued due to the limited size of L1 data cache. Such early prefetch only increases memory traffic without benefit. As stated in Section V-A, CAPS adjusts warp priority to detect the stride and the base addresses of CTAs as early as possible 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, CAPS 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 0.91% of the prefetched data was evicted from L1 by an intervening demand fetch before the prefetched data is consumed. The early prefetch ratio increases to 1.16% even without the early wake-up mechanism. Hence, for a small fractional increase of early prefetch ratio it is possible to completely avoid the early wake-up process, if desired.

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 results when CAPS is applied for the various schedulers. When CAPS is implemented on the unmodified round-robin and two-level schedulers, average distances between prefetch and demand requests are 64.3 cycles and 145.0 cycles respectively. When CAPS works cooperatively with the prefetch-aware scheduler, CAPS can issues a prefetching request on average 172.7 cycles before the target demand request. This result represents the prefetch-aware scheduler effectively improve the timeliness of CAPS.

F. Energy Consumption

Figure 15 shows energy consumption of CAPS normalized to the baseline configuration. We estimate the energy consumption of CAPS and baseline GPU with GPUWattch [32]. Power metrics of CAPS is estimated with CACTI and the synthesized RTL models based on

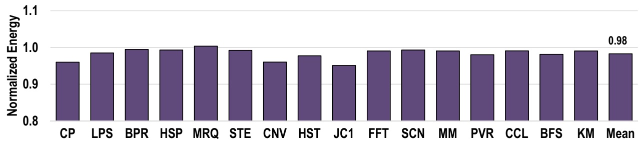


Figure 15: Energy consumption by CAPS

45 nm technology node as mentioned in Section V-D. The simulation results show CAPS consumes 2% less energy on average.

VII. Related Work

Prefetching is one of the prominent approaches to overcome the memory wall. Several studies proposed memory prefetching algorithms for GPUs.

Lee et al. [9] 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 the 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 CAPS prefetcher effectively.

Jog et al. [17] pointed out that the current rotation based warp scheduling algorithms are inefficient when working in conjunction with conventional prefetching engines. They 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 CAPS quantitatively with this approach and showed the performance improvements of CAPS.

Lakshminarayana and Kim [13] proposed a prefetching algorithm by observing a unique data access patterns in graph applications. They pointed out that the GPU 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. Whereas their work focused on prefetch for iterated load instructions in a loop that appears mostly in graph applications for GPUs, CAPS is effectively applicable to load instructions regardless of the number of iterations as far as load instructions have regular stride across warps.

Sethia et al. [16] predict prefetching addresses of a warp by expanding address trends of threads within the warp. If any two adjacent threads' addresses accessed by a load instruction have fixed distance, each thread predicts its next load address by using the distance and the total active thread count. As each thread prefetches its own next data access for each load instruction, the mechanism is highly optimized for loop-based load instructions. On the other hand, our scheme can also cover no-loop-based load instructions by predicting addresses of all the warps in a CTA. Furthermore, as we show in the example codes in Section IV, indices of data arrays are functions of CTA IDs as well as thread IDs. Hence, at the CTA boundaries one has to compute the CTA's base address at runtime.

Most of the prior art devote efforts to augment the performance impact of simple prefetchers by assuming that the pattern detection in GPU 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.

VIII. 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, detecting stride patterns across CTAs is a challenge. To tackle this challenge we propose CTA-aware prefetcher and scheduler for GPUs. CAPS 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, CAPS is able to issue timely and accurate prefetches. The evaluation results show that CAPS predicts prefetch addresses with over 97% accuracy and improves performance by 8% on average with maximum 27%.

Acknowledgment

This work was supported by DARPA-PERFECT-HR0011-12-2-0020, NSF (CNS-1705047), and Samsung Electronics.

References

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

[2] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011, pp. 235–246.

[3] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing dram latency divergence in irregular gpgpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014, pp. 128–139.

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

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

[6] ——, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[7] ——, "NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made."

[8] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in gpu," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017, pp. 307–319.

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

[10] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[11] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 34–44.

[12] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91, 1991, pp. 176–186.

[13] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, ser. HPCA '14, 2014.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[15] NVIDIA, "Cuda c programming guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide.

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

[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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, pp. 332–343.

[18] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013, pp. 395–406.

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

[20] 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 *International Symposium on Workload Characterization*, ser. IISWC '09, 2009, pp. 44–54.

[21] K. Wang and C. Lin, "Decoupled affine computation for simt gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017, pp. 295–306.

[22] G. Koo, H. Jeon, and M. Annavaram, "Revealing critical loads and hidden data locality in gpgpu applications," in *IEEE International Symposium on Workload Characterization*, ser. IISWC '15, 2015, pp. 120–129.

[23] "FreePDK process design kit," www.eda.ncsu.edu/wiki/FreePDK.

[24] S. J. E. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[25] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012, pp. 72–83.

[26] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular gpu kernels," in *IEEE International Symposium on Workload Characterization*, ser. ISSWC '14, 2014, pp. 130–139.

[27] "Parboil benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

[28] "Polybench/gpu." [Online]. Available: http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/

[29] "Scalable heterogeneous computing benchmark suite." [Online]. Available: http://keeneland.gatech.edu/software/keeneland/shoc

[30] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, 2008, pp. 260–269.

[31] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *IEEE International Symposium on Workload Characterization*, ser. IISWC '14, 2014, pp. 140–149.

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