Orchestrated Scheduling and Prefetching for GPGPUs

Adwait Jog† Onur Kayiran† Asit K. Mishra§ Mahmut T. Kandemir†
Onur Mutlu‡ Ravishankar Iyer§ Chita R. Das†
†The Pennsylvania State University ‡ Carnegie Mellon University §Intel Labs
University Park, PA 16802 Pittsburgh, PA 15213 Hillsboro, OR 97124
{adwait, onur, kandemir, das}@cse.psu.edu onur@cmu.edu {asit.k.mishra, ravishankar.iyer}@intel.com

ABSTRACT

In this paper, we present techniques that coordinate the thread scheduling and prefetching decisions in a General Purpose Graphics Processing Unit (GPGPU) architecture to better tolerate long memory latencies. We demonstrate that existing warp scheduling policies in GPGPU architectures are unable to effectively incorporate data prefetching. The main reason is that they schedule consecutive warps, which are likely to access nearby cache blocks and thus prefetch accurately for one another, back-to-back in consecutive cycles. This either 1) causes prefetches to be generated by a warp too close to the time their corresponding addresses are actually demanded by another warp, or 2) requires sophisticated prefetcher designs to correctly predict the addresses required by a future "far-ahead" warp while executing the current warp.

We propose a new *prefetch-aware* warp scheduling policy that overcomes these problems. The key idea is to separate in time the scheduling of consecutive warps such that they are not executed back-to-back. We show that this policy not only enables a simple prefetcher to be effective in tolerating memory latencies but also improves memory bank parallelism, even when prefetching is not employed. Experimental evaluations across a diverse set of applications on a 30-core simulated GPGPU platform demonstrate that the prefetch-aware warp scheduler provides 25% and 7% average performance improvement over baselines that employ prefetching in conjunction with, respectively, the commonly-employed round-robin scheduler or the recently-proposed two-level warp scheduler. Moreover, when prefetching is not employed, the prefetch-aware warp scheduler provides higher performance than both of these baseline schedulers as it better exploits memory bank parallelism.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'13 Tel Aviv, Israel
Copyright 2013 ACM 978-1-4503-2079-5/13/06 ...$15.00.

General Terms

Design, Performance

Keywords

GPGPUs; Prefetching; Warp Scheduling; Latency Tolerance

1. INTRODUCTION

The memory subsystem is a critical determinant of performance in General Purpose Graphics Processing Units (GPGPUs). And, it will become more so as more compute resources continue to get integrated into the GPGPUs and as the GPGPUs are placed onto the same chip with CPU cores and other accelerators, resulting in higher demands for memory performance.

Traditionally, GPGPUs tolerate long memory access latencies by concurrently executing many threads. These threads are grouped into fixed-sized batches known as *warps* or *wavefronts*. Threads within a warp share the same instruction stream and execute the same instruction at the same time, forming the basis for the term *single instruction multiple threads*, SIMT [3, 4, 30]. The capability to rapidly context switch between warps in the state-of-the-art GPGPUs allows the execution of other warps when one warp stalls (on a long-latency memory operation), thereby overlapping memory access latencies of different warps. The effectiveness of the *warp scheduling policy*, which determines the order and time in which different warps are executed, has a critical impact on the memory latency tolerance and memory bandwidth utilization, and thus the performance, of a GPGPU. An effective warp scheduling policy can facilitate the concurrent execution of many warps, potentially enabling all compute resources in a GPGPU to be utilized without idle cycles (assuming there are enough threads).

Unfortunately, commonly-employed warp schedulers are ineffective at tolerating long memory access latencies, and therefore lead to significant underutilization of compute resources, as shown in previous work [17, 21, 29, 34]. The commonly-used round-robin (RR) policy schedules *consecutive warps*[1] in consecutive cycles, assigning all warps equal priority in scheduling. As a result of this policy, most of the warps arrive at a long-latency memory operation roughly at the same time [17, 34]. The core can therefore become *inactive* as there may be no warps that are *not* stalling due to

[1]Two warps that have consecutive IDs are called consecutive warps. Due to the way data is usually partitioned across different warps, it is very likely that consecutive warps access nearby cache blocks [16, 17, 24, 34].

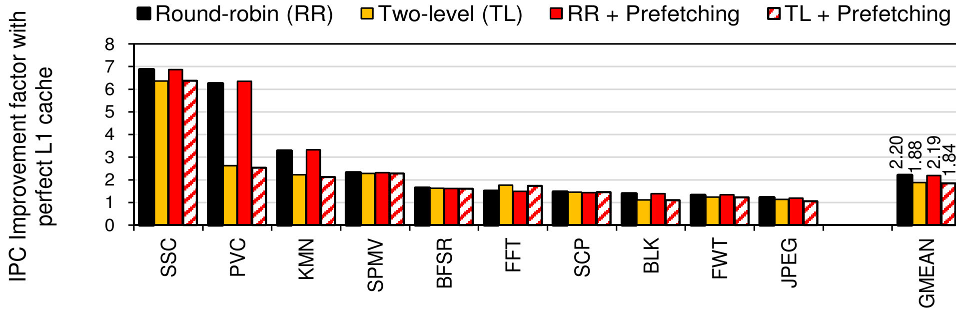


Figure 1: IPC improvement when L1 cache is made perfect on a GPGPU that employs (1) round-robin (RR) warp scheduling policy, (2) two-level (TL) warp scheduling policy, (3) data prefetching together with RR, and (4) data prefetching together with TL. Section 5 describes our evaluation methodology and workloads.

a memory operation. To overcome this disadvantage of the RR policy, the two-level (TL) warp scheduling policy [34] was proposed. This scheduler divides all warps into *fetch groups* and prioritizes warps from a single fetch group until they reach a long-latency operation. When one fetch group stalls due to a long-latency memory operation, the next fetch group is scheduled. The scheduling policy of warps within a fetch group is round robin, and so is the scheduling policy across fetch groups. The primary insight is that each fetch group reaches the long-latency operations at different points in time. As a result, when warps in one fetch group are stalled on memory, warps in another fetch group can continue to execute, thereby effectively tolerating the memory latency in a fetch group by performing computation in another.

The leftmost two bars for each application in Figure 1 show the potential performance improvement achievable if the L1 caches were perfect on 1) a GPGPU that employs the RR policy and 2) the same GPGPU but with the TL policy, across a set of ten diverse applications. The two-level warp scheduler reduces the performance impact of L1 cache misses on performance, as shown by the lower IPC improvement obtained by making the L1 data cache perfect on top of the TL policy. However, a significant performance potential remains: IPC would improve by 1.88× if the L1 cache were perfect, showing that there is significant potential for improving memory latency tolerance in GPGPU systems.

Data prefetching, commonly employed in CPU systems (e.g., [7,8,41,44]), is a fundamental latency hiding technique that can potentially improve the memory latency tolerance of GPGPUs and achieve the mentioned performance potential.[2] However, we find that employing prefetching naively does not significantly improve performance in GPGPU systems. The effect of this is shown quantitatively in the rightmost two bars for each application in Figure 1: employing a data prefetcher (based on a spatial locality detector [18], as described in detail in Section 4.2) with either the RR or the TL scheduling policy does not significantly improve performance.

In this paper, we observe that existing warp scheduling policies in GPGPUs are unable to effectively incorporate data prefetching mechanisms. The main reason is that they schedule consecutive warps, which are likely to access nearby cache blocks and thus prefetch accurately for one another, back to back in consecutive cycles. Consider the use of a simple streaming prefetcher with the RR policy: when one warp stalls and generates its demand requests, the prefetcher generates requests for the next N cache blocks. Soon after, and long before these prefetch requests complete, the succeeding warps get scheduled and likely require these cache blocks due to the high spatial locality between consecutive warps [16, 17, 24, 34]. Unfortunately, these succeeding warps cannot take advantage of the issued prefetches because the prefetch requests were issued *just before* the warps were scheduled. The TL scheduling policy suffers from the same problem: since consecutive warps within a fetch group are scheduled consecutively, the prefetches issued by a preceding warp are immediately demanded by the succeeding one, and as a result, even though prefetches are accurate, they do not provide performance improvement as they are too late. One could potentially solve this problem by designing a prefetcher that prefetches data for a "far-ahead", non-consecutive warp that will be scheduled far in the future while executing the current warp such that the prefetch requests are complete by the time the "far-ahead" warp gets scheduled. Unfortunately, this requires a more sophisticated prefetcher design: accurately predicting the addresses required by a "far-ahead", non-consecutive warp is fundamentally more difficult than accurately predicting the addresses required by the next consecutive warp due to two reasons: 1) non-consecutive warps do not exhibit high spatial locality among each other [17, 34]; in fact, the sets of addresses required by two non-consecutive warps may have no relationship with each other, 2) the time at which the far-ahead warp gets scheduled may vary depending on the other warp scheduling decisions that happen in between the scheduling of the current and the far-ahead warps.

We observe that orchestrating the warp scheduling and prefetching decisions can enable a simple prefetcher to provide effective memory latency tolerance in GPGPUs. To this end, we propose a new *prefetch-aware (PA)* warp scheduling policy. The core idea is to separate in time the scheduling of consecutive warps such that they are *not* executed back-to-back, i.e., one immediately after another. This way, when one warp stalls and generates its demand requests, a simple prefetcher can issue prefetches for the next N cache blocks, which are likely to be completed by the time the consecutive warps that need them are scheduled. While the prefetch requests are in progress, other non-consecutive warps that do not need the prefetched addresses are executed.

[2] A form of data prefetching was developed in [29] for GPGPUs, where one warp prefetches data for another.

The prefetch-aware warp scheduling policy is based on the TL scheduler, with a key difference in the way the fetch groups are formed. Instead of placing consecutive warps in the same fetch group as the TL scheduler does, the PA scheduler places *non-consecutive* warps in the same fetch group. In addition to enabling a simple prefetcher to be effective, this policy also improves memory bank-level parallelism because it enables non-consecutive warps, which are likely to access different memory banks due to the lack of spatial locality amongst each other, to generate their memory requests concurrently. Note that the PA scheduler causes a loss in row buffer locality due to the exact same reason, but the use of simple spatial prefetching can restore the row buffer locality by issuing prefetches to an already-open row.

Contributions: To our knowledge, this is the first work that coordinates thread scheduling and prefetching decisions for improving memory latency tolerance in GPGPUs. Our major contributions are as follows:

- We show that the state-of-the-art warp scheduling policies in GPGPUs are unable to effectively take advantage of data prefetching to enable better memory latency tolerance.
- We propose a *prefetch-aware* warp scheduling policy, which not only enables prefetching to be more effective in GPGPUs but also improves memory bank-level parallelism even when prefetching is not employed.
- We show that the proposed *prefetch-aware* warp scheduler can work in tandem with a simple prefetcher that uses spatial locality detection [18].
- Our experimental results show that this orchestrated scheduling and prefetching mechanism achieves 25% and 7% average IPC improvement across a diverse set of applications, over state-of-the-art baselines that employ the same prefetcher with the round-robin and two-level warp schedulers, respectively. Moreover, when prefetching is *not* employed, the proposed *prefetch-aware* warp scheduler provides respectively 20% and 4% higher IPC than these baseline schedulers as it better exploits memory bank parallelism.

2. BACKGROUND

This section provides a brief description of the baseline GPGPU architecture, typical scheduling strategies, and existing prefetching mechanisms.

Baseline GPGPU architecture: Our baseline GPGPU architecture, shown in Figure 2(A), consists of multiple simple cores, also called streaming-multiprocessors (SMs) [37]. Each core typically has a SIMT width of 8 to 32, and is associated with private L1 data, texture and constant caches, along with a shared memory. The cores are connected to memory channels (partitions) via an interconnection network. We assume that the traffic in each direction between the cores and the memory channels are serviced by two separate crossbars. Each memory partition is associated with a shared L2 cache. We assume the write-evict policy [40] for caches. Section 5 gives more details on the baseline platform configuration.

Scheduling in GPGPUs: Execution in GPGPUs starts with the launch of a kernel. In this work, we assume sequential execution of kernels, which means only one kernel is executed at a time. Each kernel consists of many *thread blocks* or *Cooperative Thread Arrays (CTAs)*. A CTA encapsulates all synchronization and barrier primitives among a group of threads [24]. The CUDA programming model places a constraint on the number of CTAs that can be concurrently executed on a core. This constraint depends on factors such as the size of the register file and the shared memory. After a kernel is launched on the GPU, the global CTA scheduler issues CTAs to the cores. In our baseline configuration, we assume a CTA allocation strategy where the CTA scheduler keeps on scheduling CTAs to an SM until the maximum CTA limit is reached. For example, in the 2-core system shown in Figure 2(B), assuming that the maximum number of concurrently executing CTAs per core is 2, if 4 CTAs need to be assigned, the first two CTAs will be assigned to core 1 and the remaining two will be assigned to core 2. In a C-core system, with a limit of N CTAs per core, if the number of unassigned CTAs is more than $N \times C$, the first core will be assigned the first set of N CTAs, the second core the second set of N CTAs, and so on. If the number of unassigned CTAs is less than $N \times C$, then less than N CTAs are assigned to each core in order to distribute the load evenly across the cores. After the assignment of the CTAs, available threads in the cores are scheduled in the core pipelines at the granularity of *warps*, where a warp typically consists of 32 threads. Every 4 cycles, a warp ready for execution is selected in a round-robin fashion and fed to the 8-way SIMT pipeline of a GPGPU core. Typically warps that have consecutive IDs (i.e., consecutive warps) have good spatial locality as they access nearby cache blocks [16,17,24,34] due to the way data is usually partitioned across different warps.

Prefetching in GPGPUs: Inter-thread L1 data prefetching [29] was recently proposed as a latency hiding technique in GPGPUs. In this technique, a group of threads prefetch data for threads that are going to be scheduled later. This inter-thread prefetcher can also be considered as an inter-warp prefetcher, as the considered baseline architecture attempts to coalesce the memory requests of all the threads in a warp as a single cache block request (e.g., 4B requests per thread × 32 threads per warp = 128B request per warp, which equals the cache block size). The authors propose that prefetching data for other warps (in turn, threads) can eliminate cold misses, as the warps for which the data is prefetched will find their requested data in the cache. In the case where the threads demand their data before the prefetched data arrives, the demand requests can be merged with the already-sent prefetch requests (if accurate) via miss status handling registers (MSHRs). In this case, the prefetch can partially hide some of the memory latency. In this work, we consider similar inter-warp prefetchers.

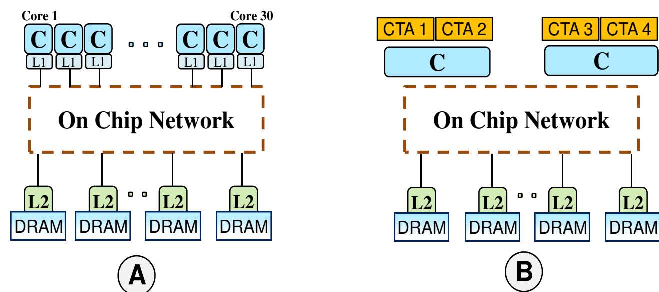


Figure 2: (A) GPGPU architecture (B) Illustrative example showing the assignment of 4 CTAs on 2 cores. Maximum CTAs/core is assumed to be 2.

3. INTERACTION OF SCHEDULING AND PREFETCHING: MOTIVATION AND BASIC IDEAS

In this section, we first illustrate the shortcomings of state-of-the-art warp scheduling policies in integrating data prefetching effectively. We then illustrate our proposal, the prefetch-aware warp scheduling policy, which aims to orchestrate scheduling and prefetching.

Our illustrations revolve around Figure 3. The left portion of the figure shows the execution and memory request timeline of a set of eight warps, W1-W8, with eight different combinations of warp scheduling and prefetching. The right portion shows the memory requests generated by these eight warps and the addresses and banks accessed by their memory requests. Note that consecutive warps W1-W4 access a set of consecutive cache blocks X, X+1, X+2, X+3, mapped to DRAM Bank 1, whereas consecutive warps W5-W8 access another set of consecutive cache blocks Y, Y+1, Y+2, Y+3, mapped to DRAM Bank 2. The legend of the figure, shown on the rightmost side, describes how different acronyms and shades should be interpreted.

3.1 Shortcomings of the State-of-the-Art Warp Schedulers

3.1.1 Round-robin (RR) warp scheduling

Figure 3 (A) shows the execution timeline of the eight warps using the commonly-used round-robin (RR) warp scheduling policy, without data prefetching employed. As described before in Section 1, since all warps make similar amounts of progress due to the round robin nature of the policy, they generate their memory requests (D1-D8) roughly at the same time, and as a result stall roughly at the same time (i.e., at the end of the first compute phase, C1, in the figure). Since there are no warps to schedule, the core remains idle until the data for at least one of the warps arrives, which initiates the second compute phase, C2, in the figure. We define the stall time between the two compute phases as *MemoryBlockCycles*. Figure 3 (A') shows the effect of RR scheduling on the DRAM system. The RR policy exploits both row buffer locality (RBL) and bank-level parallelism (BLP) in DRAM because: i) as consecutive warps W1-W4 (W5-W8) access consecutive cache blocks, their requests D1-D4 (D5-D8) access the same row in Bank 1 (Bank 2), thereby exploiting RBL, ii) warp sets W1-W4 and W5-W8 access different banks and generate their requests roughly at the same time, thereby exploiting BLP.

3.1.2 Round-robin (RR) warp scheduling and inter-warp prefetching

Figures 3 (B) and (B') show the execution timeline and DRAM state when an inter-warp prefetcher is incorporated on top of the baseline RR scheduler. The goal of the inter-warp prefetcher is to reduce *MemoryBlockCycles*. When a warp generates a memory request, the inter-warp prefetcher generates a prefetch request for the next warp (in this example, for the next sequential cache block). The prefetched data is placed in a core's private L1 data cache, which can serve the other warps. For example, the issuing of demand request D1 (to cache block X) by W1 triggers a prefetch request P1 (to cache block X+1) which will be needed by W2. Figure 3 (B) shows that, although the prefetch requests are accurate, adding prefetching on top of RR scheduling does not improve performance (i.e., reduce *MemoryBlockCycles*). This is because the next-consecutive warps get scheduled soon after the prefetch requests are issued and generate demand requests for the same cache blocks requested by the prefetcher, long before the prefetch requests are complete (e.g., W2 generates a demand request to block X+1 right after W1 generates a prefetch request for the same block). Hence, this example illustrates that RR scheduling cannot effectively take advantage of simple inter-warp prefetching.

3.1.3 Two-level (TL) warp scheduling

Figure 3 (C) shows the execution timeline of the eight warps using the recently-proposed two-level round-robin scheduler [34], which was described in Section 1. The TL scheduler forms smaller fetch groups out of the concurrently executing warps launched on a core and prioritizes warps from a single fetch group until they reach long-latency operations. The eight warps in this example are divided into two fetch groups, each containing 4 warps. The TL scheduler first executes warps in group 1 (W1-W4) until these warps generate their memory requests D1-D4 and stall. After that, the TL scheduler switches to executing warps in group 2 (W5-W8) until these warps generate their memory requests D5-D8 and stall. This policy thus overlaps some of the latency of memory requests D1-D4 with computation done in the second fetch group, thereby reducing *MemoryBlockCycles* and improving performance compared to the RR policy, as shown via *Saved cycles* in Figure 3 (C). Figure 3 (C') shows the effect of TL scheduling on the DRAM system. The TL policy exploits row buffer locality but it does not fully exploit bank-level parallelism because there are times when a bank remains idle without requests because not all warps generate memory requests at roughly the same time: during the first compute period of fetch group 2, bank 2 remains idle.

3.1.4 Two-level (TL) warp scheduling and intra-fetch-group prefetching

Figures 3 (D) and (D') show the effect of using an *intra-fetch-group prefetcher* along with the TL policy. The prefetcher used is the same inter-warp prefetcher as the one described in Section 3.1.2, where one warp generates a prefetch request for the next-consecutive warp in the same fetch group. Adding this prefetching mechanism on top of TL scheduling does not improve performance since the warp that is being prefetched for gets scheduled immediately after the prefetch is generated. This limitation is the same as what we observed when adding simple inter-warp prefetching over the RR policy in Section 3.1.2.

We conclude that prefetching for warps within the same fetch group is ineffective because such warps will be scheduled soon after the generation of prefetches.[3] Henceforth, we assume the prefetcher employed is an inter-fetch-group prefetcher.

3.1.5 Two-level (TL) warp scheduling and inter-fetch-group prefetching

The idea of an *inter-fetch-group* prefetcher is to prefetch data for the next (or a future) fetch group. There are two

[3]Note that in RR warp scheduling, although there is no fetch group formation, all the launched warps can be considered to be part of a single large fetch group.

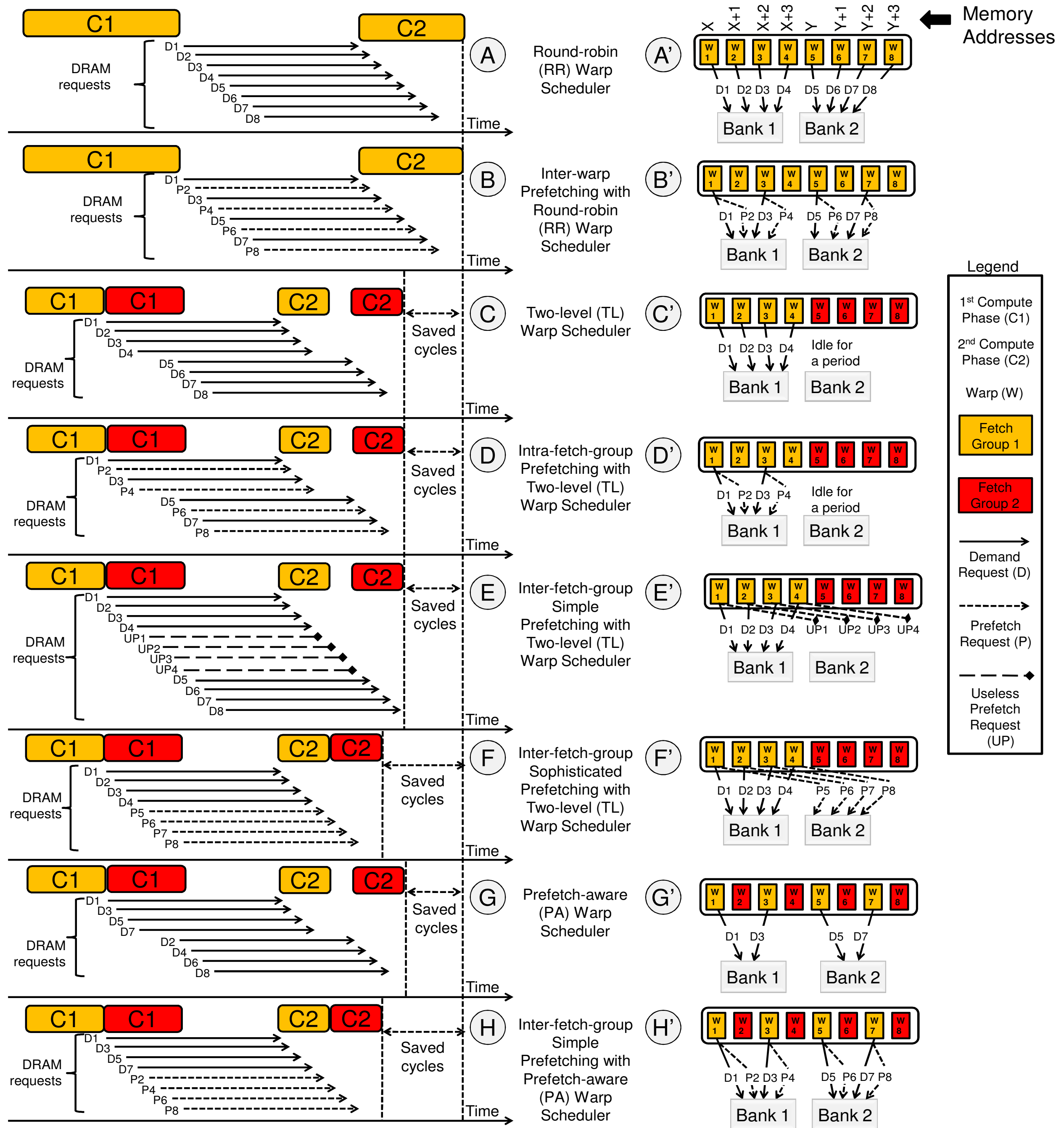


Figure 3: An illustrative example showing the working of various scheduling and prefetching mechanisms, motivating the need for the design of our prefetch-aware warp scheduler.

cases. First, if the prefetch requests are accurate, in which case a sophisticated prefetching mechanism is required as the addresses generated by the next fetch group may not have any easy-to-predict relationship to the addresses generated by the previous one, they can potentially improve performance because the prefetches would be launched long before they are needed. However, if the prefetches are inaccurate, which could be the case with a simple prefetcher, such an inter-fetch group prefetcher will issue useless prefetches.

Figures 3 (E) and (E') depict the latter case: they show the effect of using a *simple inter-fetch-group* prefetcher along with the TL policy. Since the simple prefetcher cannot accurately predict the addresses to be accessed by fetch group 2 (Y, Y+1, Y+2, Y+3) when observing the accesses made by fetch group 1 (X, X+1, X+2, X+3), it ends up issuing useless prefetches (UP1-UP4). This not only wastes valuable memory bandwidth and cache space, but may also degrade performance (although the example in the figure shows no performance loss).

Figures 3 (F) and (F') depict the former case: they show

the effect of using a *sophisticated* inter-fetch-group prefetcher along with the TL policy. Since the sophisticated prefetcher *can* accurately predict the addresses to be accessed by fetch group 2 (Y, Y+1, Y+2, Y+3) when observing the accesses made by fetch group 1 (X, X+1, X+2, X+3), it improves performance compared to the TL scheduler without prefetching. Unfortunately, as explained in Section 1, designing such a sophisticated prefetcher is fundamentally more difficult than designing a simple (e.g., next-line [12] or streaming [20]) prefetcher because non-consecutive warps (in different fetch groups) do not exhibit high spatial locality among each other [17, 34]. In fact, the sets of addresses required by two non-consecutive warps *may* have no predictable relationship with each other, potentially making such sophisticated prefetching practically impossible.

In the next two sections, we illustrate the working of our prefetch-aware warp scheduling policy which enables the benefits of a sophisticated prefetcher without requiring the implementation of one.

3.2 Orchestrating Warp Scheduling and Data Prefetching

3.2.1 Prefetch-aware (PA) warp scheduling

Figures 3 (G) and (G') show the execution timeline, group formation, and DRAM state of the eight warps using our proposed prefetch-aware (PA) scheduler. The PA policy is based on the TL scheduler (shown in Figures 3 (C and C')), with a key difference in the way the fetch groups are formed: the PA policy groups *non-consecutive* warps in the same group. Figure 3 (G') shows that warps W1, W3, W5, W7 are in fetch group 1 and warps W2, W4, W6, W8 are in fetch group 2. Similar to the TL policy, since a group is prioritized until it stalls, this policy enables the overlap of memory access latency in one fetch group with computation in another, thereby improving performance over the baseline RR policy. Different from the TL policy, the PA policy exploits bank-level parallelism but may not fully exploit row buffer locality. This is because non-consecutive warps, which do not have spatial locality, generate their memory requests roughly at the same time, and exactly because these requests do not have spatial locality, they are likely to access different banks. In this example, D1 and D3, which access the same row in Bank 1, are issued concurrently with D5 and D7, which access Bank 2, enabling both banks to be busy with requests most of the time when there are outstanding requests. However, D2 and D4, which access the same row D1 and D3 access, are issued much later, which can degrade row buffer locality if the row is closed since the time D1 and D3 accessed it. A detailed description of the PA scheduler will be provided in Section 4.1.

3.2.2 Prefetch-aware (PA) warp scheduling and inter-fetch-group prefetching

Figures 3 (H) and (H') show the execution timeline and DRAM state of the eight warps using our proposed prefetch-aware (PA) scheduler along with a simple inter-fetch-group prefetcher. The simple prefetcher is a simple next-line prefetcher, as assumed in earlier sections. Adding this prefetching mechanism on top of PA scheduling *improves* performance compared to TL scheduling, and in fact achieves the same performance as TL scheduling combined with a *sophisticated, and likely difficult-to-design* prefetcher, since the warp that is being prefetched for gets scheduled long after the warp that prefetches for it. Concretely, consecutive warps (e.g., W1 and W2) that have high spatial locality are located in different fetch groups. When the preceding warp (W1) generates its prefetch (P2 to address X+1), the succeeding warp (W2) does not get scheduled immediately afterwards but after the previous fetch group completes. As a result, there is some time distance between when the prefetch is generated and when it is needed, leading to the prefetch covering the memory access latency partially or fully, resulting in a reduction in *MemoryBlockCycles* and execution time. Figure 3 (H') also shows that using the PA policy in conjunction with a simple prefetcher fully exploits both row buffer locality and bank-level parallelism.

Conclusion: Based on this illustration, we conclude that the prefetch-aware warp scheduling policy can enable a simple prefetcher to provide significant performance improvements by ensuring that consecutive warps, which are likely to accurately prefetch for each other with a simple prefetcher, are placed in different fetch groups. In the next section, we delve into the design of the prefetch-aware warp scheduling policy and a simple prefetcher that can take advantage of this policy.

4. MECHANISM AND IMPLEMENTATION

We describe the mechanism and implementation of the *prefetch-aware* warp scheduler (Section 4.1), describe a simple spatial locality detector based prefetcher that can take advantage of it (Section 4.2), and provide a hardware overhead evaluation of both techniques (Section 4.3).

4.1 Prefetch-Aware Scheduling Mechanism

As we discussed earlier, the PA scheduler is based on the TL scheduler, but its primary difference is in the way the fetch groups are formed. The main goal of the fetch group formation algorithm is to ensure consecutive warps are *not* in the same group such that they can effectively prefetch for each other by executing far apart in time. A second goal of this algorithm is to improve memory bank-level parallelism by enabling non-consecutive warps, which do not have good spatial locality, to generate their memory requests roughly at the same time and spread them across DRAM banks. However, to enable the better exploitation of row buffer locality within a group, the algorithm we develop can assign *some* number of consecutive warps into the same group. Algorithm 1 depicts how group formation is performed in our experiments. We briefly describe its operation using an example.

Group formation depends on the number of warps available on the core (n_warps), and the number of warps in a fetch group (g_size). The number of fetch groups is equal to $\frac{n_warps}{g_size}$. To understand how fetch groups are formed, consider 32 as the maximum number of warps launched on a core, and the group size to be 8 warps. In this case, 4 fetch groups (n_grp) are formed. Warps are enumerated from 0 to 31, and fetch groups are enumerated from 0 to 3. W0 (warp 0) is always assigned to G0 (group 0). The 8th (as group size is equal to 8) warp (W8) is also assigned to G0. Similarly, W16 and W24 are assigned to G0, in a modular fashion until we reach W31. Since G0 has only 4 warps, 4 more warps need to be assigned to G0. The modular assign-

ment procedure continues with the first unassigned warp, which is W1 in this case, and places it in G0 along with W9, W17 and W25. Note that, in this example, two consecutive warps belong to the same fetch group, e.g., G0 contains both W0 and W1. The number of consecutive warps in a fetch group, n_cons_warps, is equal to $\lfloor \frac{g_size}{n_grp} \rfloor$. Having placed 8 warps in G0, in order to form G1, the algorithm starts from W2. In a similar manner, first, W2, W10, W18 and W26, and then, W3, W11, W19 and W27 are assigned to G1. The group assignment policy exemplified above can be formulated by $g_num[i] = \lfloor \frac{i \bmod g_size}{n_cons_warps} \rfloor$, where $g_num[i]$ denotes the group number of warp i. If there are no consecutive warps in the same group, the above formula simplifies to $g_num[i] = i \bmod g_size$. Algorithm 1 more formally depicts how our evaluated PA scheduler forms fetch groups.[4] Note that, in our evaluations, we used a fetch group size of 8.[5] The number of warps on the cores depends on the application and the programming model, and is limited by the core resources.

Algorithm 1 Fetch group formation in the PA scheduler

```
                        ▷ warp and fetch group numbers start from 0
  ▷ n_warp is the number of concurrently-executing warps on a core
                    ▷ g_size is the number of warps in a fetch group
                          ▷ n_warp is assumed to be divisible by g_size
                                  ▷ n_grp is the number of fetch groups
  ▷ n_cons_warps is the number of consecutive warps in a group.
Its minimum value 1.
                    ▷ g_num[i] is the fetch group number of warp i

procedure FORM_GROUPS(n_warp, g_size)
    n_grp ← n_warp / g_size
    n_cons_warps ← ⌊ g_size / n_grp ⌋
    for i = 0 → n_warp − 1 do
        g_num[i] ← ⌊ (i mod g_size) / n_cons_warps ⌋
    end for
    return g_num
end procedure
```

4.2 Spatial Locality Detection Based Prefetching

We develop a *simple* prefetching algorithm that tries to prefetch for consecutive warps (which belong to different fetch groups in the PA scheduler). The key idea of the algorithm is to first detect the regions of memory that are frequently accessed (i.e., *hot* regions), and based on this information predict the addresses that are likely to be requested soon.

Spatial locality detection: In order to detect the frequently-accessed memory regions, we use a spatial locality detector (SLD) similar to the one used in [18]. This technique involves the tracking of cache misses that are mapped to the same *macro-block*, which is defined as a group of consecutive cache blocks. In our evaluations, the size of a macro-block is 512 bytes (i.e., 4 cache blocks). SLD maintains a fixed number of macro-block entries in a per-core SLD table (which is organized as a fully-associative 64-entry table in our evaluations). After a main memory request is generated, its macro-block address is searched in the SLD table. The SLD table entry records, using a bit vector, which cache blocks in the macro-block have already been requested. If no matching entry is found, the least-recently-used entry is replaced, a new entry is created, and the corresponding bit in the bit vector is set. If a matching entry is found, simply the corresponding bit in the bit vector is set. The number of bits that are set in the bit vector indicate the number of unique cache misses to the macro-block.

Prefetching mechanism: The key idea is to issue prefetch requests for the cache lines in the *hot* macro-block that have not yet been demanded. Our prefetching mechanism considers a macro-block hot if at least C cache blocks in the macro-block were requested. We set C to 2 in our experiments. For example, if the macro-block size is 4 cache blocks and C is 2, as soon as the number of misses to the macro-block in the SLD table reaches 2, the prefetcher issues prefetch requests for the remaining two cache blocks belonging to that macro-block. Note that this SLD based prefetcher is relatively conservative, as it has a prefetch degree of 2 and a low prefetch distance, because it is optimized to maximize accuracy and minimize memory bandwidth wastage in a large number of applications running on a GPGPU where memory bandwidth is at premium.

Analysis: We analyze the effect of warp scheduling, the TL and PA schedulers in particular, on macro-block access patterns. Figure 4 shows the distribution of main memory requests, averaged across all fetch groups, that are to macro-blocks that have experienced, respectively, 1, 2, and 3-4 cache misses. Figure 4 (a) shows this distribution averaged across all applications, (b) shows it on `PVC`, and (c) shows it on `BFSR`. These two applications are chosen as they show representative access patterns – `PVC` exhibits high spatial locality, `BFSR` does not. Several observations are in order. First, with the TL scheduler, on average across all applications, 36% of memory requests of a fetch group access all cache blocks of a particular macro-block. This means that 36% of the requests from a fetch group have *good* spatial locality. This confirms the claim that the warps in a fetch group have good spatial locality when the TL scheduler is used. However, this percentage goes down to 17% in the PA scheduler. This is intuitive because the PA scheduler favors non-consecutive warps to be in the same fetch group. This results in a reduction in spatial locality between memory requests of a fetch group, but on the flip side, spatial locality can be regained by issuing prefetch requests (as explained in Section 3.2.2) to the unrequested cache blocks in the macro-block. Second, 38% of the memory requests from a fetch group access only one cache block of a particular macro-block with the TL scheduler. In `BFSR`, the percentage of such requests goes up to 85%. If an application has a high percentage of such requests, macro-block prefetching is not useful, because these request patterns do not exhibit high spatial locality.

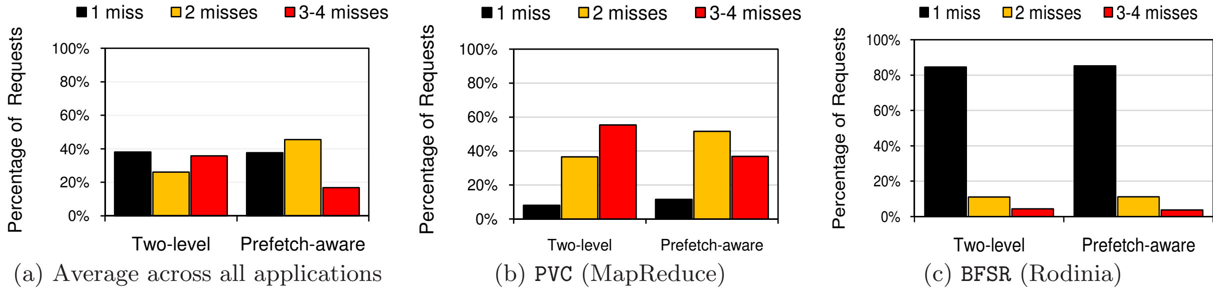
4.3 Hardware Overhead

We evaluate the hardware overhead of the PA scheduler and the spatial locality based prefetcher. We implemented the two mechanisms in RTL using Verilog HDL and synthesized them using the Synopsys Design Compiler on 65nm TSMC libraries [43].

Scheduling: Lindholm et al. [30] suggest that the warp scheduler used in NVIDIA GPUs has zero-cycle overhead, and warps can be scheduled according to their pre-

[4]Note that the group formation we used in Figure 3 is for illustrative purposes only and does not strictly follow this algorithm.

[5]Past works [17, 34] that developed scheduling algorithms that form groups showed that a group size of 8 provides the best performance on experimental setups similar to ours.



(a) Average across all applications (b) PVC (MapReduce) (c) BFSR (Rodinia)

Figure 4: The distribution of main memory requests, averaged across all fetch groups, that are to macro-blocks that have experienced, respectively, 1, 2, and 3-4 unique cache misses. Section 5 describes our methodology and workloads.

Table 1: Simulated baseline GPGPU configuration

Core Configuration	1300MHz, SIMT width = 8
Resources / Core	Max. 1024 threads (32 warps, 32 threads/warp), 32KB shared memory, 32684 registers
Caches / Core	32KB 8-way L1 data cache, 8KB 4-way texture cache, 8KB 4-way constant cache, 128B cache block size
L2 Cache	16-way 128 KB/memory channel, 128B cache block size
Default Warp Scheduling	Round-robin warp scheduling (among ready warps)
Advanced Warp Scheduling	Two-level warp scheduling [34] (fetch group size = 8 warps)
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Interconnect	1 crossbar/direction (30 cores, 8 MCs), concentration = 3, 650MHz
Memory Model	8 GDDR3 Memory Controllers (MC), FR-FCFS scheduling (64 max. requests/MC), 8 DRAM-banks/MC, 2KB row size, 1107 MHz memory clock
GDDR3 Timing [10]	$t_{CL} = 10$, $t_{RP} = 10$, $t_{RC} = 35$, $t_{RAS} = 25$, $t_{RCD} = 12$, $t_{RRD} = 8$, $t_{CDLR} = 6$, $t_{WR} = 11$

determined priorities. Since the difference between PA and TL schedulers is primarily in the fetch group formation approach, the hardware overhead of our proposal is similar to that of the TL scheduler. The implementation of the PA scheduler requires assignment of appropriate scheduling priorities (discussed in Section 4.1) to all warps on a core. We synthesized the RTL design of the PA scheduler and found that it occupies 814 μm^2 on each core.

Prefetching: We implement the spatial locality detection based prefetcher as described in Section 4.2. We model a 64-entry SLD table and 4 cache blocks per macro-block. This design requires 0.041 mm^2 area per core. Note that the prefetcher is not on the critical path of execution.

Overall Hardware Overhead: For a 30-core system, the required hardware occupies 1.25 mm^2 chip area. This overhead, calculated using a 65nm design, corresponds to 0.27% of the area of the Nvidia GTX 285, which is also a 30-core system, yet is implemented in a smaller, 55nm process technology.

5. EVALUATION METHODOLOGY

We evaluate our schemes using an extensively modified GPGPU-Sim 2.1.2b [2], a cycle-accurate GPGPU simulator. Table 1 provides the details of the simulated platform. We study 10 CUDA applications derived from representative application suites (shown in Table 2), where thread-level parallelism is not enough to hide long memory access latencies. We run these applications until they complete their execution or reach 1B instructions, whichever comes first.

In addition to using *instructions per cycle (IPC)* as the primary performance metric for evaluation, we also consider auxiliary metrics such as bank-level parallelism (BLP) and row buffer locality (RBL). BLP is defined as the average number of memory banks that are accessed when there is at least one outstanding memory request at any of the banks [22, 23, 32, 33]. Improving BLP enables better utilization of DRAM bandwidth. RBL is defined as the average hit-rate of the row buffer across all memory banks [23]. Improving RBL increases the memory service rate and also enables better DRAM bandwidth utilization. We also measure the L1 data cache miss rates when prefetching is employed. Accurate and timely prefetches can lead to a reduction in miss rates.

Table 2: Evaluated GPGPU applications

#	Suite	Application	Abbr.
1	MapReduce [14, 17]	SimilarityScore	SSC
2	MapReduce [14, 17]	PageViewCount	PVC
3	Rodinia [5]	Kmeans Clustering	KMN
4	Parboil [42]	Sparse Matrix Multiplication	SPMV
5	Rodinia [5]	Breadth First Search	BFSR
6	Parboil [42]	Fast Fourier Transform	FFT
7	CUDA SDK [36]	Scalar Product	SCP
8	CUDA SDK [36]	Blackscholes	BLK
9	CUDA SDK [36]	Fast Walsh Transform	FWT
10	Third Party	JPEG Decoding	JPEG

6. EXPERIMENTAL RESULTS

Figure 5 shows the IPC improvement of five different combinations of warp scheduling and prefetching normalized to the baseline RR scheduler: 1) RR scheduler with data prefetching, 2) TL scheduler, 3) TL scheduler with spatial locality detection based prefetching, 4) PA scheduler, 5) PA scheduler with prefetching. Overall, without prefetching, the PA scheduler provides 20% average IPC improvement over the RR scheduler and 4% over the TL scheduler. When prefetching is employed, the PA scheduler provides 25% improvement over the RR scheduler and 7% over the TL scheduler, leading to performance within 1.74× of a perfect L1 cache. The rest of this section analyzes the different

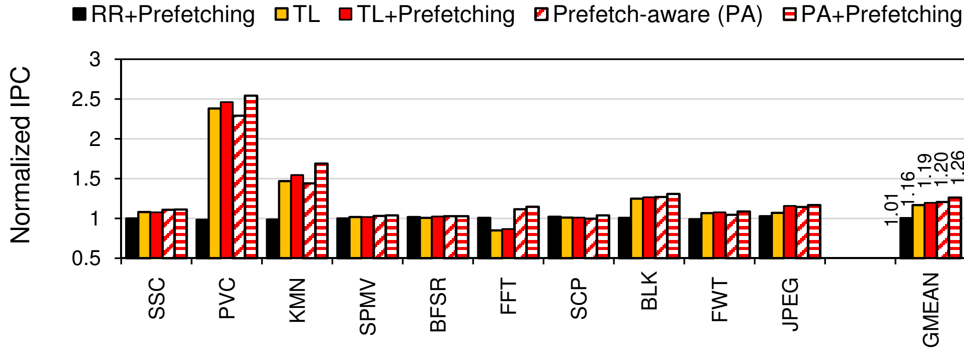
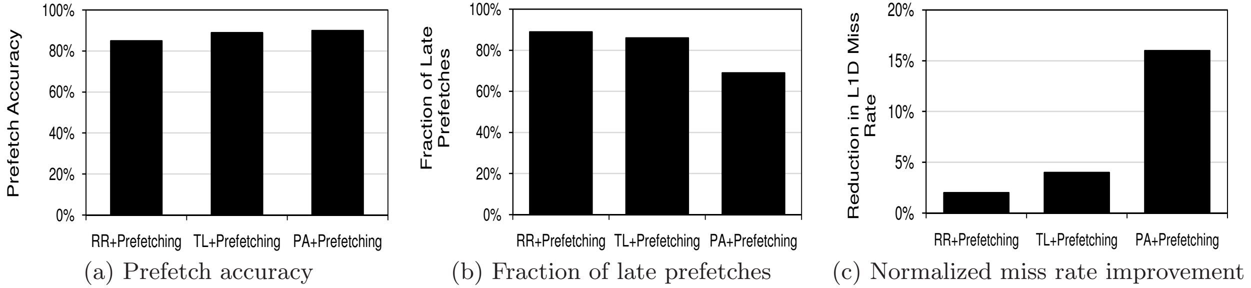


Figure 5: IPC performance impact of different scheduling and prefetching strategies. Results are normalized to the IPC with the RR scheduler.



(a) Prefetch accuracy (b) Fraction of late prefetches (c) Normalized miss rate improvement

Figure 6: (a) Prefetch accuracy, (b) Fraction of late prefetches, and (c) Reduction in L1 data cache miss rate when prefetching is implemented with each scheduler. The results are averaged across all applications.

combinations of scheduling and prefetching.

Effect of prefetching with the RR scheduler: On average, adding our SLD-based prefetcher over the RR scheduler provides only 1% IPC improvement over the RR scheduler without prefetching. The primary reason is that the prefetcher cannot lead to timely transfer of data as described in Section 3.1.2. Figure 6 shows that even though 85% of the issued prefetches are accurate with the RR scheduler, 89% of these accurate prefetches are late (i.e., are needed before they are complete), and as a result the prefetcher leads to an L1 data cache miss reduction of only 2%.

Effect of the TL scheduler: As discussed in Section 3.1.3, the TL scheduler improves latency tolerance by overlapping memory stall times of some fetch groups with computation in other fetch groups. This leads to a 16% IPC improvement over the RR scheduler due to better core utilization. One of the primary limitations of TL scheduling is its inability to maximize DRAM bank-level parallelism (BLP). Figure 7 shows the impact of various scheduling strategies on BLP. The TL scheduler leads to an average 8% loss in BLP over the RR scheduler. In `FFT`, this percentage goes up to 25%, leading to a 15% performance loss over the RR scheduler.

Effect of prefetching with the TL scheduler: Figure 8 shows the L1 data cache miss rates when prefetching is incorporated on top of the TL scheduler. Using the spatial locality detection based prefetcher decreases the L1 miss rate by 4% over the TL scheduler without prefetching. In `KMN` and `JPEG`, this reduction is 15% and 2%, respectively. This L1 miss rate reduction leads to a 3% performance improvement over the TL scheduler. Note that the performance improvement provided by prefetching on top of the TL scheduler is relatively low because the issued prefetches are most of the time too late, as described in Section 3.1.4. However, the performance improvement of prefetching on top of the TL scheduler is still higher than that on top of the RR scheduler because the prefetcher we use is not *too* simple (e.g., not as simple as a next-line prefetcher) and its combination with the TL scheduler enables some inter-fetch-group prefetching to happen. Figure 6 shows that 89% of the issued prefetches are accurate with the TL scheduler. 85% of these accurate prefetches are still late, and as a result the prefetcher leads to an L1 data cache miss reduction of only 4%.

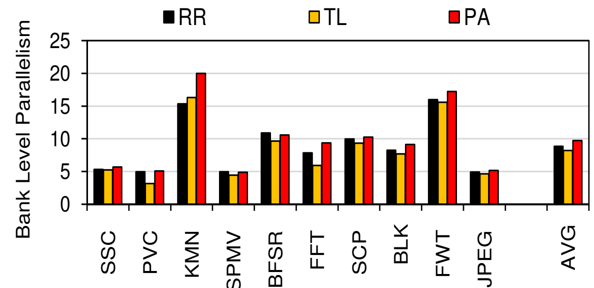


Figure 7: Effect of various scheduling strategies on DRAM bank-level parallelism (BLP)

Effect of the PA scheduler: As described in Section 3.2.1, the PA scheduler is likely to provide high bank-level parallelism at the expense of row buffer locality because it concurrently executes non-consecutive warps, which are likely to access different memory banks, in the same fetch group. Figures 7 and 9 confirm this hypothesis: on average, the PA scheduler improves BLP by 18% over the TL sched-

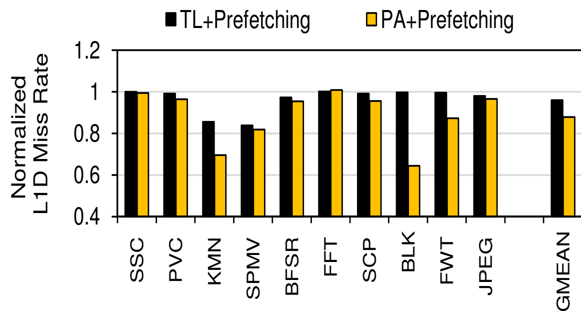


Figure 8: Effect of various scheduling and prefetching strategies on L1D miss rate. Results are normalized to miss rates with the TL scheduler.

uler while it degrades RBL by 24%. The PA scheduler is expected to work well in applications where the loss in row locality is less important for performance than the gain in bank parallelism. For example, the PA scheduler provides a 31% increase in IPC (see Figure 5) over the TL scheduler in `FFT` due to a 57% increase in BLP, even though there is a 44% decrease in row locality compared to the TL scheduler. On the contrary, in `PVC`, there is 4% reduction in IPC compared to TL scheduler due to a 31% reduction in row-locality, even though there is a 62% increase in BLP. This shows that both row locality and bank-level parallelism are important for GPGPU applications, which is in line with the observations made in [15] and [17]. On average, the PA scheduler provides 20% IPC improvement over the RR scheduler and 4% improvement over the TL scheduler.

Effect of prefetching with the PA scheduler: The use of prefetching together with the proposed PA scheduler provides two complementary benefits: (1) the PA scheduler enables prefetching to become more effective by ensuring that consecutive warps that can effectively prefetch for each other get scheduled far apart in time from each other, (2) prefetching restores the row buffer locality loss due to the use of the PA scheduler. We evaluate both sources of benefits in Figures 8 and 9, respectively: (1) the use of prefetching together with the PA scheduler reduces the L1 miss rate by 10% compared to the TL scheduler with prefetching, (2) the addition of prefetching over the PA scheduler improves RBL such that the RBL of the resulting system is within 16% of the RBL of the TL scheduler with prefetching. Figure 6 shows that the PA scheduler significantly improves prefetch timeliness compared to the TL and RR schedulers while also slightly increasing prefetch accuracy: with the PA scheduler, 90% of the prefetches are accurate, of which only 69% are late (as opposed to the 85% late prefetch fraction with the TL prefetcher). Overall, our orchestrated spatial prefetching and prefetch-aware scheduling mechanism improves performance by 25% over the RR scheduler with prefetching and by 7% over the TL scheduler with prefetching (as seen in Figure 5). We conclude that the new prefetch-aware warp scheduler effectively enables latency tolerance benefits from a simple spatial prefetcher.

Case analyses of workload behavior: In `KMN`, prefetching with the PA scheduler provides 10% IPC improvement over prefetching with the TL scheduler on account of a 31% decrease in L1 miss rate and 6% increase in row locality over the PA scheduler. Note that in `KMN`, prefetching on top of the PA scheduler leads to the restoration of 100% of the row-locality lost by the PA scheduler over the TL scheduler. In `SPMV`, the use of prefetching improves row buffer locality by 2× over the TL scheduler. This enhancement, along with a 9% increase in BLP over the TL scheduler leads to 3% higher IPC over TL+Prefetching. In `BFSR`, we do not observe any significant change in performance with our proposal because of the high number of unrelated memory requests that do not have significant spatial locality: in this application, many macro-blocks have only one cache block accessed, as was shown in Figure 4 (c). In `FFT`, adding prefetching on top of the PA scheduler provides 3% additional performance benefit. Yet, as described earlier, `FFT` benefits most from the improved BLP provided by the PA scheduler. In `FFT`, the combination of prefetching and PA scheduling actually *increases* the L1 miss rate by 1% compared to the combination of prefetching and TL scheduling (due to higher cache pollution), yet the former combination has 3% higher performance than the latter as it has much higher bank-level parallelism. This shows that improving memory bank-level parallelism, and thereby reducing the *cost* of each cache miss by increasing the overlap between misses can actually be more important than reducing the L1 cache miss rate, as was also observed previously for CPU workloads [38].

Analysis of cache pollution due to prefetching: One of the drawbacks of prefetching is the potential increase in cache pollution, which triggers early evictions of cache blocks that are going to be needed later. To measure cache pollution, we calculated the Evicted Block Reference Rate (EBRR) using Equation 1. This metric indicates the fraction of read misses that are to cache blocks that were in the cache but were evicted due to a conflict.

$$EBRR = \frac{\#read\ misses\ to\ already\ evicted\ cache\ blocks}{\#read\ misses} \quad (1)$$

Figure 10 shows that prefetching causes a 26% increase in EBRR when a 32KB L1 data cache (as in our evaluations) is used. When cache size is increased to 64KB, the increase in EBRR goes down to 10%. This is intuitive as a large cache will have fewer conflict misses. One can thus reduce pollution by increasing the size of L1 caches (or by incorporating prefetch buffers), but that would lead to reduced hardware resources dedicated for computation, thereby hampering thread-level parallelism and the ability of the architecture to hide memory latency via thread-level parallelism. Cache pollution/contention can also be reduced via more intelligent warp scheduling techniques, as was shown in prior work [17, 40]. We leave the development of warp scheduling mechanisms that can reduce pollution in the presence of prefetching as a part of future work.

7. RELATED WORK

To our knowledge, this is the first paper that shows that the commonly-employed GPGPU warp scheduling policies are *prefetch-unaware*, and hence straightforward incorporation of simple prefetching techniques do not lead to large performance benefits. The key contribution of this paper is a prefetch-aware warp scheduler, which not only enhances the effectiveness of prefetching but also improves overall DRAM bandwidth utilization. We briefly describe and compare to the closely related works.

Scheduling techniques in GPUs: The two-level warp

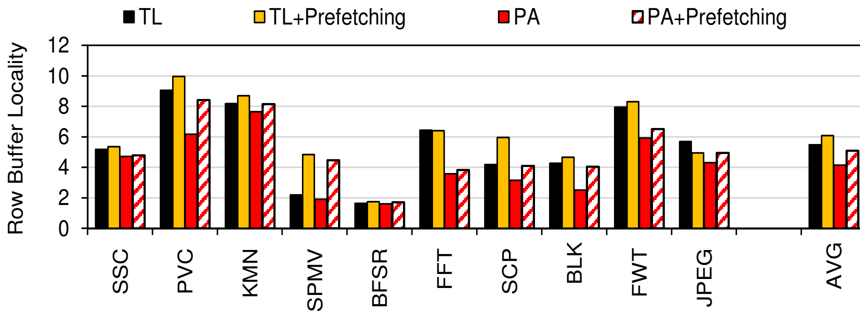


Figure 9: Effect of different scheduling and prefetching strategies on DRAM row buffer locality

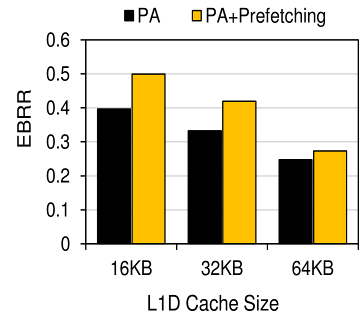


Figure 10: Effect of prefetching on Evicted Block Reference Rate (EBRR) for various L1 data cache sizes

scheduler proposed by Narasiman et al. [34] splits the concurrently executing warps into groups to improve memory latency tolerance. We have already provided extensive qualitative and quantitative comparisons of our proposal to the two-level scheduler. Rogers et al. [40] propose cache-conscious wavefront scheduling to improve the performance of cache-sensitive GPGPU applications. Gebhart et al. [11] propose a two-level warp scheduling technique that aims to reduce energy consumption in GPUs. Kayiran et al. [21] propose a CTA scheduling mechanism that dynamically estimates the amount of thread-level parallelism to improve GPGPU performance by reducing cache and DRAM contention. Jog et al. [17] propose OWL, a series of CTA-aware scheduling techniques to reduce cache contention and improve DRAM performance for bandwidth-limited GPGPU applications. None of these works consider the effects of warp scheduling on data prefetching. Our work examines the interaction of scheduling and prefetching, and develops a new technique to orchestrate these two methods of latency tolerance.

Data prefetching: Lee et al. [29] propose a many-thread aware prefetching strategy in the context of GPGPUs. Our prefetch-aware warp scheduling technique can be synergistically combined with this prefetcher for better performance. Jog et al. [17] propose a memory-side prefetching technique that improves L2 cache hit rates in GPGPU applications. Our work describes how a spatial locality detection mechanism can be used to perform core-side data prefetching. Lee et al. [28] evaluate the benefits and limitations of both software and hardware prefetching mechanisms for emerging high-end processor systems. Many other prefetching and prefetch control mechanisms (e.g., [6,9,19,35,41]) have been proposed within the context of CPU systems. Our prefetch-aware scheduler is complementary to these techniques. We also provide a specific core-side prefetching mechanism for GPGPUs that is based on spatial locality detection [18].

Row buffer locality and bank-level parallelism: Several memory request scheduling techniques for improving bank-level parallelism and row buffer locality [1, 22, 23, 25–27,31,33,39,45] have been proposed. In particular, the work by Hassan et al. [13] quantifies the trade-off between BLP and row locality for multi-core systems, and concludes that bank-level parallelism is more important. Our results show that the prefetch-aware warp scheduler, which favors bank-level parallelism, provides higher average performance than the two-level scheduler [34], which favors row buffer locality (but this effect could be due to the characteristics of applications we evaluate). On the other hand, Jeong et al. [15] observe that both bank-level parallelism and row buffer locality are important in multi-core systems. We also find that improving row locality at the expense of bank parallelism improves performance in some applications yet reduces performance in others, as evidenced by the two-level scheduler outperforming the prefetch-aware scheduler in some benchmarks and vice versa in others. Lakshminarayana et al. [25] propose a DRAM scheduling policy that essentially chooses between Shortest Job First and FR-FCFS [39, 46] scheduling policies at run-time, based on the number of requests from each thread and their potential of generating a row buffer hit. Yuan et al. [45] propose an arbitration mechanism in the interconnection network to restore the lost row buffer locality caused by the interleaving of requests in the network when an in-order DRAM request scheduler is used. Ausavarungnirun et al. [1] propose a staged memory scheduler that batches memory requests going to the same row to improve row locality while also employing a simple in-order request scheduler at the DRAM banks. Lee et al. [26, 27] explore the effects of prefetching on row buffer locality and bank-level parallelism in a CPU system, and develop memory request scheduling [26, 27] and memory buffer management [27] techniques to improve both RBL and BLP in the presence of prefetching. Mutlu and Moscibroda [31, 33] develop mechanisms that preserve and improve bank-level parallelism of threads in the presence of inter-thread interference in a multi-core system. None of these works propose a *warp scheduling* technique that exploits bank-level parallelism, which our work does. We explore the interplay between row locality and bank parallelism in GPGPUs, especially in the presence of prefetching, and aim to achieve high levels of both by intelligently orchestrating warp scheduling and prefetching.

8. CONCLUSION

This paper shows that state-of-the-art thread scheduling techniques in GPGPUs cannot effectively integrate data prefetching. The main reason is that *consecutive thread warps*, which are likely to generate accurate prefetches for each other as they have good spatial locality, are scheduled closeby in time with each other. This gives the prefetcher little time to hide the memory access latency before the ad-

dress prefetched by one warp is requested by another warp.

To orchestrate thread scheduling and prefetching decisions, this paper introduces a *prefetch-aware (PA) warp scheduling* technique. The main idea is to form groups of thread warps such that those that have good spatial locality are in separate groups. Since warps in different thread groups are scheduled at separate times, *not* immediately after each other, this scheduling policy enables the prefetcher to have more time to hide the memory latency. This scheduling policy also better exploits memory bank-level parallelism, even when employed without prefetching, as threads in the same group are more likely to spread their memory requests across memory banks.

Experimental evaluations show that the proposed prefetch-aware warp scheduling policy improves performance compared to two state-of-the-art scheduling policies, when employed with or without a hardware prefetcher that is based on spatial locality detection. We conclude that orchestrating thread scheduling and data prefetching decisions in a GPGPU architecture via prefetch-aware warp scheduling can provide a promising way to improve memory latency tolerance in GPGPU architectures.

Acknowledgments

We thank the anonymous reviewers, Nachiappan Chidambaram Nachiappan, and Bikash Sharma for their feedback on earlier drafts of this paper. This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687, #0953246, and grants from Intel, Microsoft and Nvidia. Onur Mutlu is partially supported by an Intel Early Career Faculty Award.

References

[1] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.
[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.
[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC/ETAPS 2010*.
[4] M. Bauer, H. Cook, and B. Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *SC*, 2011.
[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.
[6] D. Chiou, S. Devadas, J. Jacobs, P. Jain, V. Lee, E. Peserico, P. Portante, L. Rudolph, G. E. Suh, and D. Willenson. Scheduler-Based Prefetching for Multilevel Memories. Technical Report Memo 444, MIT, July 2001.
[7] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. Technical report, Intel Corporation, 2006.
[8] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *MICRO*, 2009.
[9] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for Bandwidth-efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *HPCA*, 2009.
[10] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *MICRO*, 2011.
[11] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. In *ISCA*, 2011.
[12] J. D. Gindele. Buffer Block Prefetching Method. *IBM Technical Disclosure Bulletin*, 20(2), July 1977.
[13] S. Hassan, D. Choudhary, M. Rasquinha, and S. Yalamanchili. Regulating Locality vs. Parallelism Tradeoffs in Multiple Memory Controller Environments. In *PACT*, 2011.
[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.
[15] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems . In *HPCA*, 2012.
[16] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *ICS*, 2012.
[17] A. Jog, O. Kayiran, N. C. 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 *ASPLOS*, 2013.
[18] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-Time Spatial Locality Detection and Optimization. In *MICRO*, 1997.
[19] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *ISCA*, 1997.
[20] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *ISCA*, 1990.
[21] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *CSE Penn State Tech Report, TR-CSE-2012-006*, 2012.
[22] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.
[23] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.
[24] D. Kirk and W. W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.
[25] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. *Computer Architecture Letters*, 2012.
[26] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controllers. In *MICRO*, 2008.
[27] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving Memory Bank-level Parallelism in the Presence of Prefetching. In *MICRO*, 2009.
[28] J. Lee, H. Kim, and R. Vuduc. When Prefetching Works, When It Doesn't, and Why. In *TACO*, 2012.
[29] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *MICRO*, 2010.
[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2), 2008.
[31] T. Moscibroda and O. Mutlu. Distributed Order Scheduling and Its Application to Multi-core Dram Controllers. In *PODC*, 2008.
[32] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.
[33] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.
[34] 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 *MICRO*, 2011.
[35] K. Nesbit and J. Smith. Data Cache Prefetching Using a Global History Buffer. In *HPCA*, 2004.
[36] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.
[37] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, Nov. 2011.
[38] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. ISCA, 2006.
[39] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.
[40] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.
[41] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA*, 2007.
[42] J. A. Stratton et al. Scheduler-Based Prefetching for Multilevel Memories. Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012.
[43] Synopsys Inc. *Design Compiler*.
[44] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM J. Res. Dev.*, Jan. 2002.
[45] G. Yuan, A. Bakhoda, and T. Aamodt. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *MICRO*, 2009.
[46] W. K. Zuravleff and T. Robinson. Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order. (U.S. Patent Number 5,630,096), Sept. 1997.