Barrier-Aware Warp Scheduling for Throughput Processors

Yuxi Liu[§†‡], Zhibin Yu[†], Lieven Eeckhout[‡], Vijay Janapa Reddi[¶],
Yingwei Luo[§], Xiaolin Wang[§], Zhenlin Wang[*], Chengzhong Xu[†‖]

[§]Peking University [†]Shenzhen Institute of Advanced Technology, CAS [‡]Ghent University
[¶]University of Texas at Austin [*]Michigan Tech University [‖]Wayne State University

ABSTRACT

Parallel GPGPU applications rely on barrier synchronization to align thread block activity. Few prior work has studied and characterized barrier synchronization within a thread block and its impact on performance. In this paper, we find that barriers cause substantial stall cycles in barrier-intensive GPGPU applications although GPGPUs employ lightweight hardware-support barriers. To help investigate the reasons, we define the execution between two adjacent barriers of a thread block as a *warp-phase*. We find that the execution progress within a warp-phase varies dramatically across warps, which we call *warp-phase-divergence*. While warp-phase-divergence may result from execution time disparity among warps due to differences in application code or input, and/or shared resource contention, we also pinpoint that warp-phase-divergence may result from warp scheduling.

To mitigate barrier induced stall cycle inefficiency, we propose *barrier-aware warp scheduling (BAWS)*. It combines two techniques to improve the performance of barrier-intensive GPGPU applications. The first technique, most-waiting-first (MWF), assigns a higher scheduling priority to the warps of a thread block that has a larger number of warps waiting at a barrier. The second technique, critical-fetch-first (CFF), fetches instructions from the warp to be issued by MWF in the next cycle. To evaluate the efficiency of BAWS, we consider 13 barrier-intensive GPGPU applications, and we report that BAWS speeds up performance by 17% and 9% on average (and up to 35% and 30%) over loosely-round-robin (LRR) and greedy-then-oldest (GTO) warp scheduling, respectively. We compare BAWS against recent concurrent work SAWS, finding that BAWS outperforms SAWS by 7% on average and up to 27%. For non-barrier-intensive workloads, we demonstrate that BAWS is performance-neutral compared to GTO and SAWS, while improving performance by 5.7% on average (and up to 22%) compared to LRR. BAWS' hardware cost is limited to 6 bytes per streaming multiprocessor (SM).

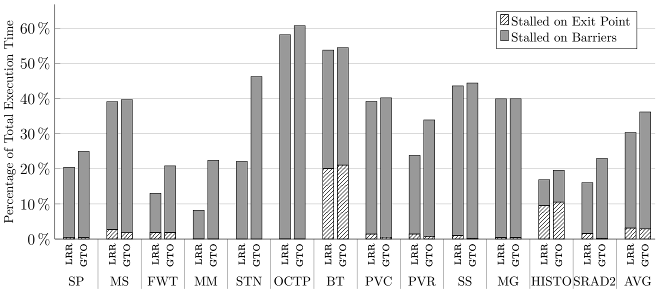

Figure 1: Fraction of time that a warp is stalled on barriers for our set of barrier-intensive GPGPU applications; computed as the average across all warps.

1. INTRODUCTION

GPGPU programming models have emerged as an important computational paradigm, allowing programmers to leverage hundreds of thousands of threads to achieve massive computational power. Programming models such as CUDA [1], ATI Stream Technology [2], and OpenCL [3] make it easy to leverage the graphics hardware to perform general-purpose parallel computing, so-called GPGPU computing. As developers increasingly utilize the hardware for a variety of different applications, all aspects of GPU hardware execution efficiency are being stressed to the limits.

In this paper, we focus on the efficiency of barrier synchronization. GPGPU applications often need synchronization to guarantee correct execution. Therefore, GPGPUs provide lightweight hardware-support barriers to implement synchronization between warps within a thread block (TB) via shared memory in the streaming multiprocessor (SM). Barriers partition TB execution into multiple phases, which we refer to as a *warp-phase*. GPGPUs schedule a new TB to execute only after all warps of the previous TB on the same SM have finished their execution, adding an internal barrier at the end point of each TB [31]. Previous studies have observed that the execution of warps within a TB could finish at very different times [19, 31], called *warp-level-divergence*. A very recent, concurrent work called SAWS [21] addresses the synchronization issue between multiple warp schedulers but not between warps within warp-phases. We focus on execution time disparity within warp-phases, at a finer granularity, that we hereby call *warp-phase-divergence*.

We discover that warp-phase-divergence leads to significant performance degradation in barrier-intensive applications. As our simulation results in Figure 1 show, the fraction of the time stalled by a warp as a result of barriers (including the exit points

[1] Corresponding author: Zhibin Yu (zb.yu@siat.ac.cn).

[2] Most of the work was done at the Shenzhen Institute of Advanced Technology, CAS.

ICS '16, June 01-03, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926267

of TBs) can range up to 61% with an average of 30% and 37% for the Loosely-Round-Robin (LRR) [1] and Greedy-then-Oldest (GTO) [26] warp scheduling policies, respectively.

No prior work, to the best of our knowledge, has comprehensively characterized and studied how barrier synchronization at the warp-phase level affects performance. We therefore analyze warp-phase-divergence and we find there are several sources of the problem: application code, input data, shared resource contention, and warp scheduling policy. Application code and input data may cause warps to execute along different paths, which may result in execution time disparity. Shared resource contention may also lead to execution time disparity, even though warps are executing the same code. A much less well understood source of warp-phase-divergence is the warp scheduling policy, which may lead to dramatically different execution rates as warp scheduling may prioritize some warps over others.

Unfortunately, commonly used and previously proposed warp scheduling policies, such as prefetch-aware [14] and memory-aware scheduling [12, 13, 23, 26], do not take barrier behavior into account. As a result, the warp scheduler does not schedule the slowest warp in a warp-phase to execute with the highest priority, which may lead to dramatic execution disparity between warps in a TB and cause substantial stall cycles during the barrier of that warp-phase. In addition, existing warp schedulers focus only on choosing warps in a better way at the issue stage of a GPGPU SIMD pipeline, but leave the warp selection at the instruction fetch stage as is in a round-robin fashion. However, with barriers, the instruction fetch unit may fetch an instruction for a warp that is waiting at a barrier while an instruction needed by another critical warp is not fetched, further exacerbating the stalls.

To address the barrier-induced stall cycle inefficiency at warp-phase granularity, we propose *barrier-aware warp scheduling (BAWS)*. This new policy is a hybrid approach that combines two online techniques. The key idea of our first technique, called *most-waiting-first (MWF)*, is based on the following insight: by scheduling the warp in a TB that has the largest number of other warps waiting for it at a barrier to execute first, we can reduce the number of stall cycles before that barrier. Our second technique, called *critical-fetch-first (CFF)*, builds on top of MWF and fetches the instruction for the warp that is the most likely to be issued by MWF in the next cycle. CFF eliminates the idle cycles caused by the warp scheduling mismatch between the fetch and issue stages in a GPGPU pipeline, specifically in the context of barriers.

In summary, the contributions of this paper are as follows:

- We show that barriers cause significant performance degradation in barrier-intensive GPGPU workloads, and we perform an in-depth and comprehensive characterization of GPGPU barrier execution.

- We mitigate barrier stall cycle inefficiency by introducing two new warp scheduling algorithms, named MWF for the issue stage and CFF for the fetch stage of a GPGPU SIMD pipeline. MWF (most-waiting-first) chooses a warp that has the largest number of other warps waiting for it to execute first. CFF (critical-fetch-first) first fetches the instruction for a warp that is the most likely to be issued by MWF in the next cycle.

- We combine MWF and CFF to form the barrier-aware warp scheduling (BAWS) policy and evaluate it by using 13 barrier-intensive GPGPU benchmarks. The results show that BAWS (MWF+CFF) improves performance for the experimented benchmarks by 17% and 9% on average (and up to 35% and 30%) over LRR [1] and GTO [26], respectively. BAWS outperforms SAWS [21] by 7% on average and up to 27%. Moreover, for non-barrier-intensive workloads, we demonstrate that BAWS is performance-neutral compared to GTO and SAWS, while improving performance by 5.7% on average (and up to 22%) compared to LRR. Hardware required for implementing BAWS is limited to 6 bytes per SM.

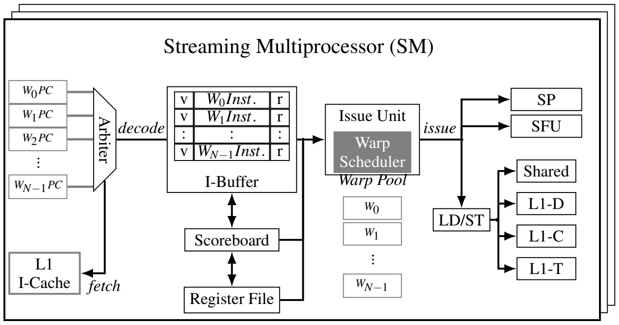

Figure 2: Microarchitecture of an SM.

The paper is organized as follows. Section 2 describes the background about the SM microarchitecture and two warp scheduling algorithms: LRR and GTO. Section 3 characterizes the barrier behavior in GPGPU workloads. Section 4 describes our barrier-aware warp scheduling (BAWS) policy. Section 5 depicts our experimental setup. Section 6 provides the evaluation results and analysis. Section 7 describes related work, and Section 8 concludes the paper.

2. BACKGROUND

GPGPU hardware employs a three-level hierarchical architecture: a streaming processor (SP) which logically executes a warp; a streaming multiprocessor (SM) consisting of many SPs, which physically executes warps in parallel; and a GPGPU processor containing a number of SMs, which runs a grid of TBs. We describe the SM microarchitecture in detail because warp scheduling occurs at the individual SM level. We subsequently provide an overview of two existing and most widely used warp scheduling algorithms: loosely-round-robin (LRR) and greedy-then-oldest (GTO).

2.1 SM Microarchitecture

The microarchitecture of a single GPGPU core is composed of a scalar front-end (fetch, decode, issue) and a SIMD back-end, as illustrated in Figure 2. Following NVIDIA's use of terminology, a SIMD back-end consists of many SIMD lanes known as Streaming Processor (SP), Special Function Unit (SFU), and Load/Store unit (LD/ST). A GPGPU core named as Streaming Multi-processor (SM) by NVIDIA typically has 32 SPs that share a single scalar front-end. Since we focus on the warp scheduling policy at the front-end, we focus on more details of the fetch unit, and simplify the illustration of the back-end structure in Figure 2.

There are 6 important hardware structures in the front-end of a GPGPU SIMD pipeline: the instruction cache (I-Cache), instruction buffer (I-Buffer), fetch, branch, decode, and issue unit. The fetch unit fetches instructions from the I-Cache according to program counters (PC). The I-Buffer serves as a temporary instruction station after an instruction is decoded but before it is issued to execute. Each warp has a fixed number of slots (e.g., 2) in the I-Buffer. Each I-Buffer slot contains a *v*-bit indicating whether an instruction is present and an *r*-bit indicating that it is ready for execution. Inside the issue unit, there is a warp scheduler selecting warps to execute according to a certain warp scheduling algorithm. Based on this microarchitecture, GPGPUs employ the *Single Instruction Multiple Thread* (SIMT) execution model in which multiple threads are grouped together to form a *warp* or *wavefront* executing in a lock-step manner.

2.2 Scheduling for the Fetch and Issue Stage

At the fetch stage, a warp is eligible for instruction fetching if it does not have any valid instructions within its I-Buffer slots. Typically, eligible warps are scheduled to fetch instructions from the I-Cache in a round-robin manner. At the issue stage, the warp scheduler selects a warp with a ready instruction in the I-Buffer according to a certain scheduling algorithm to issue. Once an instruction has been issued, its corresponding *v*-bit is altered to invalid. Then the fetch unit is signaled that it may fetch the next instruction for this warp.

The commonly used warp scheduling algorithm for the issue stage is *loosely-round-robin (LRR)* [1]. The LRR warp scheduling algorithm treats warps equally and issues them in a round-robin manner. If a warp cannot be issued, the next warp in round-robin order is to be issued. Although LRR considers fairness between warps, it does not take warp-specific features such as long memory latency into account. To address this issue, a lot of memory-aware warp scheduling algorithms have been proposed [12, 13, 14, 23, 26].

Amongst the various algorithms, the *greedy-then-oldest (GTO)* warp scheduling algorithm [26] generally outperforms the other ones for most GPGPU benchmarks. We provide a brief description about GTO because we compare against it in this paper. GTO runs a warp until it stalls, and then picks the oldest ready warp. The age of a warp is determined by the time it is assigned to the SM. However, GPGPUs assign warps to an SM by the granularity of a thread block (TB) and hence all warps in a TB have the same age. In such a case, the warps are prioritized by warp-*id*: the smaller warp-*id* gets the higher priority.

Typically, LRR suffers more from structural hazards than GTO. In each cycle, LRR selects a new warp to execute in a round-robin fashion. Since GPGPUs are based on the SIMT computing paradigm and a warp consists of a fixed number of threads, it is highly possible that the current warp executes the same instruction as that of the previous warp. In addition, most GPGPU computing operations such as ADD take several cycles to complete. As such, adjacent warps contend severely for the same compute resources, such as the SP and SFU units, and warps with a larger *id* have to wait. In contrast, GTO lets an SM greedily execute the instructions in a single warp. As such, GPGPU performance benefits from two aspects: (1) better instruction and data locality of threads and increased cache hit rate; and (2) less resource contention between warps.

3. BARRIER CHARACTERIZATION

The goal in this paper is to mitigate barrier induced stall cycle inefficiency. Before introducing our barrier-aware warp scheduling policy in the next section, we now first define terminology and characterize barrier behavior with a specific focus on barrier imbalance due to warp scheduling.

In a barrier-intensive GPGPU application, the execution lifetime of a warp can be divided into multiple *phases*, called *warp-phases*. For example, Figure 3 illustrates the execution of the 8 warps in TB0 running on SM0 for the BT (B+ Tree) benchmark. The lifetime of the 8 warps is divided into 5 warp-phases by 4 explicit barriers and 1 implicit one that corresponds to the exit point for TB0.

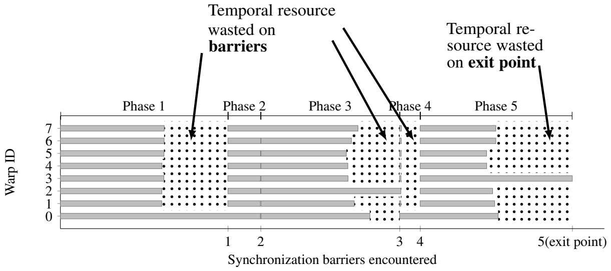

Figure 3: Warp-phases for TB0 on SM0 for benchmark BT**(B+ Tree).**

The execution progress of the warps before a barrier may be dramatically different (as is the case for barriers 1, 3, 4, and 5), but it could also be (approximately) the same (barrier 2). Xiang et al. [31] and Lee et al. [19] observe similar execution disparity before the exit point of a TB (barrier 5) and call it *warp-level-divergence*. In Figure 3, we observe that divergence also happens within a finer-grained warp-phase, and we thus refer to this new observation as *warp-phase-divergence*.

There are a number of sources of warp-phase-divergence. One source is code-dependent disparity: different threads execute different code paths (i.e., the code being executed depends on the thread-*id*), which may lead to execution time disparity among warps. Another source is input-dependent disparity: different threads, although executing the same code path, may also lead to execution time disparity among warps as they work on different input data (i.e., branch divergence triggered by data values). Yet another source of execution time disparity among warps is resource contention in shared resources such as cache and main memory. While these sources of warp-phase-divergence are quite intuitive, we focus here on an additional source of warp-phase-divergence induced by the warp scheduling algorithm itself, which is much less well understood.

3.1 Warp Scheduling Disparity

Warp scheduling may induce warp-phase-divergence by prioritizing some warps over others at the warp-phase level. For example, GTO always runs a single warp until it stalls because of its greedy nature; GTO then prioritizes the warp with the smallest warp-*id* over other warps. Figure 4 (left) shows the execution of the first three TBs scheduled by GTO for the FWT benchmark as an example. Each TB is executing 16 warps. According to GTO, the warps in TB0 are scheduled to execute first since they have the oldest age (highest priority). For the warps in the same TB which have the same priority, GTO schedules the warp with the smallest warp-*id* to execute first upon a stall. As a result, the warps in TB0 execute faster than those in TB1 and much faster than those in TB2. In particular, in TB2, warp-47 is the slowest warp, upon which warp-32 (which arrived at the barrier first) has been waiting for approximately 3,800 cycles (see warp-phase 1 in TB2). In addition, there are many stall cycles in other warp-phases such as warp-phases 3 and 5 in TB1, and warp-phase 2 in TB2. Although the number of stall cycles in these warp-phases is smaller than for warp-phase 1 in TB2, it still accumulates to a significant number of stall cycles.

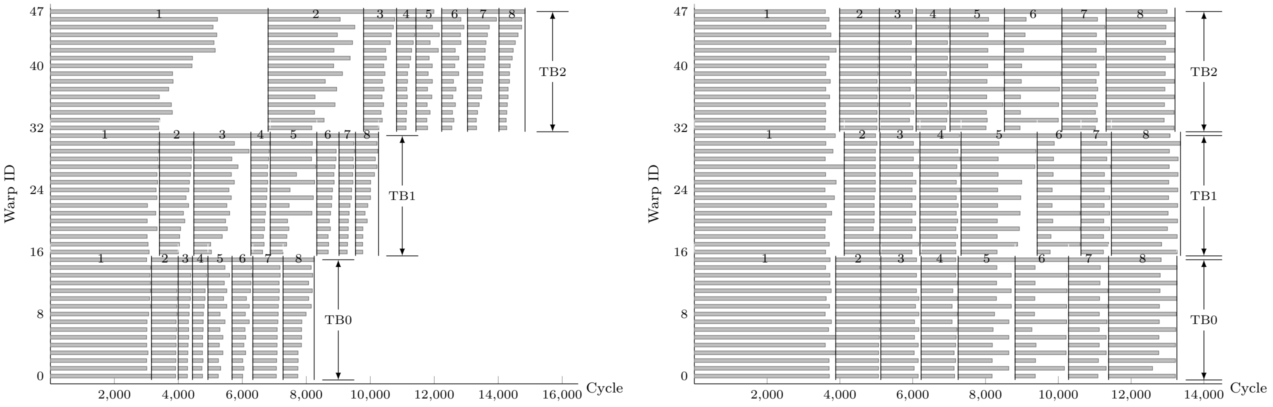

Figure 4: Execution of TB0, 1 and 2 on SM0 for FWT **under GTO (left) and LRR (right); there are 16 warps per TB; the numbers represent warp-phase *id*s.**

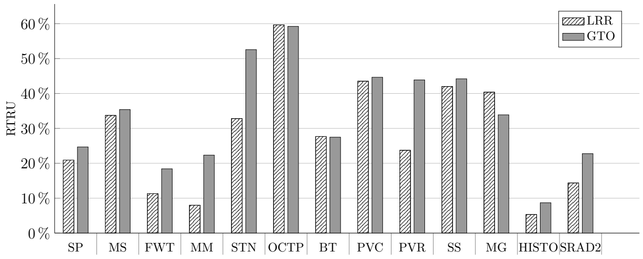

Figure 5: RTRU under LRR and GTO.

It is interesting to compare the execution behavior under GTO, see Figure 4 (left), against the execution behavior under LRR, see Figure 4 (right), again for the first three TBs for FWT. Clearly, LRR achieves a much more balanced execution compared to GTO, as LRR selects warps in a round-robin fashion. In other words, GTO leads to increased warp execution disparity. Nevertheless, GTO outperforms LRR: TBs 1 and 2 finish much earlier under GTO than under LRR, which enables other TBs to execute earlier (not shown in the Figure). The reason why GTO outperforms LRR is because it reduces the number of structural hazards, as we will quantify later in this paper. The reduction in structural hazards outpaces the increase in warp execution disparity, which ultimately leads GTO to outperform LRR.

3.2 RTRU Characterization

To systematically quantify the number of stall cycles due to warp scheduling disparity, we use the *ratio of temporal resource underutilization (RTRU)* metric defined by Xiang et al. [31]. RTRU was originally proposed for quantifying execution disparity at TB exit points. We adopt this definition to quantify disparity at the warp-phase level as follows:

$$RTRU = \frac{\sum_i (maxT - T_i)}{N \cdot maxT}, \qquad (1)$$

with $maxT$ the longest execution time of all warps, T_i the execution time of warp i, and N the number of warps. As such, RTRU indicates the execution imbalance of warps within a warp phase. Smaller is better: a small RTRU indicates less imbalance, whereas a high RTRU indicates high imbalance.

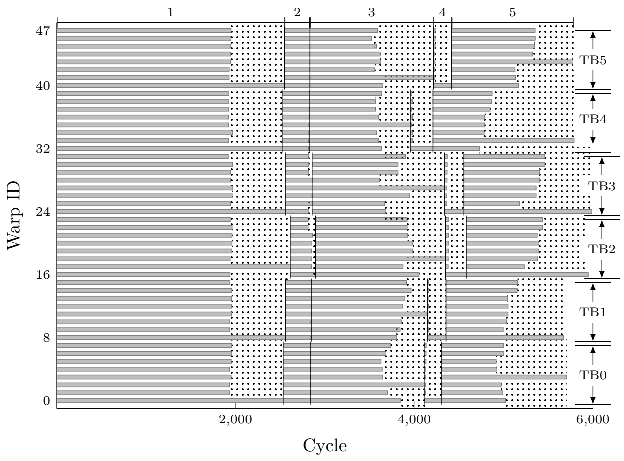

Figure 6: Warp-phases for TBs 0 through 5 for the BT **benchmark.**

Figure 5 quantifies RTRU for all of our barrier-intensive benchmarks under LRR and GTO. It is remarkable that the execution imbalance among warps in a warp phase is substantial for these barrier-intensive benchmarks. For several benchmarks we report RTRU values in the 20% to 50% range, and up to 60%. It is also interesting to note that the imbalance is typically larger under GTO than under LRR, most notably for FWT, MM, STN and PVR.

The data implies that warp scheduling may exacerbate the warp-phase-divergence problem inherent to a workload. Note though, again, that GTO typically outperforms LRR by reducing the number of structural hazards, as we will quantify later in this paper. The goal hence is to improve performance beyond GTO by reducing warp execution disparity while not introducing structural hazards, as LRR does.

3.3 Critical Warps

Before presenting barrier-aware scheduling (BAWS), we first want to illustrate the need for an online mechanism. Warp-phase-divergence leads to substantial execution time disparity at small time scales, as argued before. Some warps may reach the barrier before others do, turning the lagging warps to be *critical warps*, i.e., the thread block is waiting for the critical warp(s) to also reach the barrier before proceeding to the next warp-

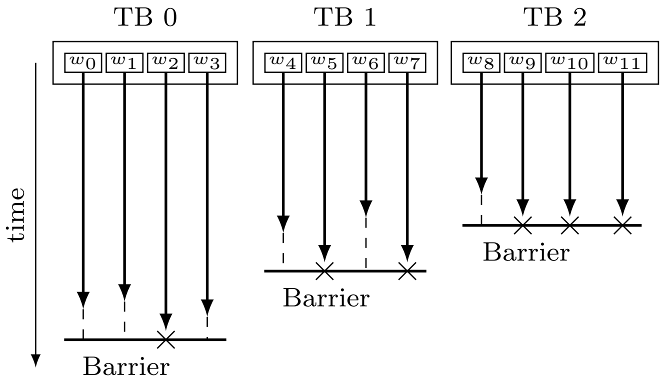

Figure 7: Overview of Most-Waiting-First (MWF) warp scheduling at the issue stage. We assume that there are 3 TBs running on a single SM concurrently and each TB contains 4 warps.

phase. Critical warp(s) may vary over time. This is illustrated in Figure 6 for the BT benchmark: the critical warp in TB0 varies from warp 0 (in phase 1), warp 2 (in phase 3), warp 0 again (in phase 4), to warp 3 (in phase 5). This illustrates that we need an online mechanism to dynamically identify critical warps to accelerate.

4. BARRIER-AWARE WARP SCHEDULING

Barrier-aware warp scheduling (BAWS) consists of two warp scheduling algorithms for the issue and fetch stage, respectively. The warp scheduling algorithm for the issue stage is called *most-waiting-first (MWF)*; the one for the fetch stage is named *critical-fetch-first (CFF)*.

4.1 Most-Waiting-First Warp Scheduling

The insight behind MWF is to schedule warps in a TB that have the largest number of other warps waiting for it at a barrier to execute first. In doing so, we improve performance by reducing the number of stall cycles before barriers.

4.1.1 Mechanism

MWF is an online technique that assigns priorities to warps at the moment when there is at least one warp that is at a barrier. MWF assigns the highest priority to the warps in a TB that has the largest number of warps waiting at a barrier. These warps are called *critical warps*. Warps in the same TB have the same priority. If two or more TBs have the same number of barrier-waiting warps, MWF assigns the highest priority to the warps in the TB with the smallest TB-*id*, similar to what GTO does for warps.

We consider two variants to MWF with respect to scheduling warps (with the same priority) within a TB. The *MWF(LRR)* variant employs LRR within a TB, and starts from the warp next (in round robin order) to the one which was issued in the last cycle in the TB. The second *MWF(GTO)* variant employs GTO, and starts from the warp issued in the last cycle, and if that one is stalled, it selects the oldest warp in the TB (i.e., with the smallest warp-*id*) to go next. We will evaluate both variants of MWF in the evaluation section.

Figure 7 shows how MWF scheduling works. Six warps from three TBs have arrived at their respective barriers, waiting for the other six warps. The number of stalled warps because of barriers for TB0, TB1, and TB2 are 1, 2, and 3, respectively. MWF assigns the highest priority 3 to the warps in TB2, a lower priority 2 to those in TB1, and the lowest priority 1 to those in TB0. For the TBs (TB0 and TB1) that have more than one warp that are still running, MWF employs either (i) LRR to schedule these warps starting from the warp next (in round robin order) to the one which was issued in the last cycle — the *MWF(LRR)* variant — or (ii) GTO to schedule these warps from the one issued in the last cycle and then oldest — the *MWF(GTO)* variant.

index = TB-*id*	priority
0	1
1	2
2	3

Table 1: The warp priority table (WPT).

For example, consider MWF(LRR) and assume the warp issued in the last cycle is $w7$ in TB1 and it reaches the barrier in the current cycle, MWF would schedule $w4$ to execute because it is the next warp to $w7$ in round-robin order. For TB0, suppose the warp lastly issued is $w0$, MWF would schedule $w1$ to execute. As a result, the scheduling orders of the 6 running warps of the three TBs in the current cycle are $w8, w4, w6, w1, w3$ and $w0$. For the warps in two or more TBs which have the same priority, suppose there is another TB (TB3, not shown in Figure 7) that also has two warps waiting at a barrier, then warps in TB3 have the same priority as those in TB1. In this case, MWF assigns higher priority to the warps in TB1 because its *id* is the smallest between TB1 and TB3.

Note that MWF naturally handles multiple warp schedulers per SM if we maintain one shared list of TB priorities. TBs with the highest priority will be prioritized by the respective warp schedulers.

4.1.2 Hardware Support

Hardware support for synchronization is available in modern GPGPUs [21]. A global synchronization control unit (SCU) keeps track of the barrier status for all warps within each TB and all TBs within an SM. The unit keeps track whether a warp is waiting on a barrier, and raises a signal once a barrier is cleared. BAWS requires a small addition to the SCU. We propose the *warp priority table (WPT)* to maintain the warp priorities on an SM. The WPT has as many rows as there are TBs, with each entry in the table containing the TB's priority, or the number of warps in the TB that have arrived at the barrier, see Table 1. A TB's priority is a counter that is incremented when one of its warps reaches the barrier; and is reset when all warps have reached the barrier. In terms of hardware cost, the WPT needs '#TBs per SM' entries, with each entry being a counter to represent 'max. #warps per TB'. Assuming 8 TBs per SM and max. 48 warps per TB (6-bit priority counters), this leads to a total WPT hardware cost of 48 bits (6 bytes) per SM.

4.2 Critical-Fetch-First Warp Scheduling

As mentioned before, the fetch unit of an SM typically fetches instructions from the I-Cache in a round-robin manner. The instructions fetched are then stored in the I-Buffer, as shown in Figure 2. When the warp scheduler in the issue unit selects a warp to issue, it first checks the I-Buffer to see whether the next instruction for that warp has been fetched. The implicit assumption here is that the warp scheduling algorithm used in the fetch stage orchestrates well with the scheduler in the issue stage. However, the warp scheduling algorithm in the fetch

Benchmark	#TBs per SM	#warps per TB
Scalar Product (SP) [24]	3	16
Merge Sort (MS) [24]	6	8
Fast Walsh Transform (FWT) [24]	3	16
Matrix Multiply (MM) [24]	6	8
Stencil (STN) [29]	2	16
Octree Partitioning (OCTP) [5]	4	8
B+ tree (BT) [6]	5	8
Page View Count (PVC) [11]	6	8
Page View Rank (PVR) [11]	6	8
Similarity Score (SS) [11]	6	8
MRI Cartesian Gridding (MG) [30]	3	16
Histogram (HISTO) [30]	3	16
Speckle Reducing Anisotropic Diffusion (SRAD2) [6]	6	8

Table 2: The barrier-intensive GPGPU benchmarks used in this study, including their characteristics (number of TBs per SM, and number of warps per TB).

stage usually does not match well with the scheduler in the issue stage, which leads to delayed execution of critical warps in barrier-intensive applications.

To address this issue, we propose *critical-fetch-first (CFF)* warp scheduling for the fetch stage. The instruction of the warp that will be issued in the next cycle is defined as the critical instruction which should be fetched. CFF therefore makes the arbiter (see Figure 2) in the fetch unit select the warp selected by MWF and fetches its instruction to its I-Buffer slots. As such, the CFF can match the pace of MWF for the issue stage, reducing fetch stall cycles.

While we propose CFF to work in concert with MWF, CFF by itself could also be employed in conjunction with other warp scheduling policies. In particular, for GTO, CFF will fetch an instruction from the warp that was currently selected at the issue stage, or if that one is stalled, it fetches an instruction from the oldest warp (smallest warp-*id*).

5. EXPERIMENTAL SETUP

We use GPGPU-sim v3.2.2 [4] for all our experiments. GPGPU-sim is a cycle-level performance simulator that models a general-purpose GPU microarchitecture. The simulator is configured to simulate NVIDIA's Fermi GTX480. It has 15 SMs, each with a warp size of 32 threads with a SIMD width of 32. With 32,768 registers per SM, each SM can support 1,536 threads. Each SM also has a 16 KB D-Cache and 2 KB I-Cache. It has 2 SP units and 1 SFU unit. There are two schedulers per SM, an even and odd scheduler that concurrently execute even and odd warps. The two schedulers operate independently from each other.

We consider 13 barrier-intensive GPGPU applications from three GPGPU benchmark suites: CUDA SDK [24], Rodinia [6], and Parboil [30]. We also employ some barrier-intensives benchmarks used in recent papers including [11, 29]. We analyze all workloads in these benchmark suites, and label a GPGPU workload to be barrier-intensive when barriers cause significant performance degradation. More precisely, we define a workload as a barrier-intensive benchmark when the fraction of stall cycles of a warp due to barriers exceeds 15% of the total execution time. The benchmarks are listed in Table 2. The second and third columns show the number of TBs concurrently running on an SM and the number of warps per TB, respectively.

6. EVALUATION

We now evaluate barrier-aware warp scheduling (including its variants), and compare its performance against previously proposed warp scheduling policies. We analyze where the performance improvement comes from by quantifying the stall latency distribution breakdown. We finally evaluate how BAWS affects the performance of non-barrier-intensive GPGPU benchmarks.

6.1 Performance Evaluation

In BAWS, MWF is the primary warp scheduling algorithm and CFF is the assistant one. We therefore evaluate them together and we use LRR as our baseline warp scheduling algorithm. We define speedup as follows:

$$speedup = \frac{IPC_{ws}}{IPC_{LRR}}, \quad (2)$$

with IPC_{ws} the IPC of warp scheduling algorithm ws. In this work, we consider the following scheduling algorithm ws:

1. TLS: Two-Level Scheduling [8];
2. GTO: Greedy-Then-Oldest [26];
3. MWF(LRR): Most-Waiting-First with LRR within a TB; and
4. MWF(GTO): Most-Waiting-First with GTO within a TB.

We consider all of these warp scheduling algorithms in two variants, with and without CFF, leading to eight scheduling policies in total.

Figure 8 quantifies the speedup of these eight warp scheduling algorithms over the baseline LRR policy. There are a number of interesting observations and conclusions to be made from these results.

BAWS outperforms LRR and GTO by a significant margin. We report that BAWS outperforms LRR by a significant margin: MWF (LRR and GTO variants) improve performance by 9.5% and 9.8% on average. MWF+CFF does even better, improving performance by 16.2% for the LRR variant and 16.9% for the GTO variant. This is significantly better than GTO, which improves performance by merely 8.4% over LRR; MWF+CFF improves upon GTO by 7.8% on average for the LRR variant and 8.5% for the GTO variant.

GTO may cause performance anomalies, BAWS does not. It is worth noting that BAWS never degrades performance over LRR, whereas GTO does, see SRAD2. BAWS achieves 30% higher performance than GTO for SRAD2. The reason is that the critical warp for this benchmark is the last warp of the last TB. As GTO schedules warps with smaller *id*s first in a greedy manner, the last warp from the last TB has to stall the longest time, severely degrading performance. BAWS on the other hand is able to identify this critical warp as soon as one warp in the last TB reaches the barrier; it then assigns a higher priority to that critical warp, improving overall performance.

CFF improves performance significantly for MWF. It is interesting to note that CFF and MWF work well in concert. Whereas CFF only slightly improves performance for TLS and GTO (by 1.7% on average), CFF has a much bigger impact on performance for MWF, improving performance by 6.7% for the LRR variant and 7.1% for the GTO variant. The intuition is that MWF

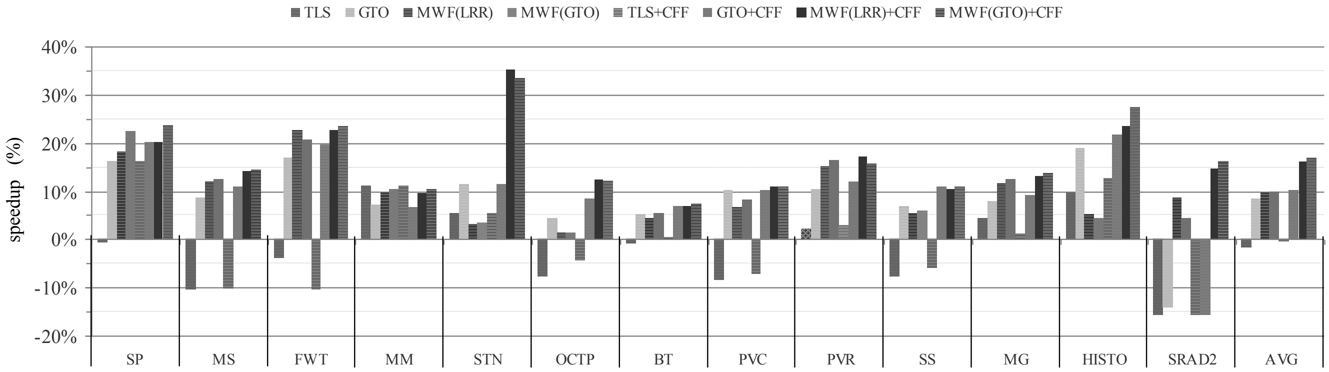

Figure 8: Speedup (percentage performance improvement) over the baseline warp scheduling algorithm LRR.

identifies the critical warp at each barrier, and prioritizes this critical warp for issuing. However, if the fetch scheduler was unable to keep up with the issue scheduler, and there are no instructions in the I-Buffer ready to be issued for the critical warp, then no forward progress can be made for the critical warp at the issue stage.

MWF requires CFF support to be effective. MWF (both the LRR and GTO variants) only slightly outperforms GTO (by 1.1% to 1.4% on average) without CFF support. MWF significantly improves performance over GTO for half the benchmarks, while being performance-neutral for a couple. Surprisingly, there are a number of benchmarks for which MWF severely degrades performance compared to GTO. This is most notably the case for the STN and HISTO benchmarks, and the OCTP and PVC benchmarks to a lesser extent. While MWF is able to accurately identify the critical warps at barriers, it suffers from the fetch stage not being able to keep up with the issue stage, as previously discussed. By engaging CFF to overcome the mismatch between the fetch and issue stages, we observe that BAWS improves performance over GTO for all benchmarks by a significant margin, by 7.8% to 8.5% on average.

TLS is performance-neutral, in contrast to BAWS. Traditional two-level scheduling (TLS) policies implement two levels of scheduling to better hide long-latency instructions. BAWS implements two levels of scheduling with the top-level scheduler at the TB-level (MWF) and the second-level one within a TB (LRR/GTO). TLS degrades performance for a number of barrier-intensive benchmarks by 5% to 15%, and is performance-neutral on average compared to LRR. BAWS on the other hand, improves performance for all benchmarks, demonstrating that two levels of scheduling are best implemented at the TB-level (first level) and within a TB (second level) to balance execution in the presence of barriers.

6.2 Fetch policy: CFF vs. LRR

One of the conclusions from the previous section is that CFF is critical for MWF to be effective. In other words, in spite of the fact that MWF is able to accurately identify the critical warp in the context of barrier synchronization, the fact that the I-Buffer lacks instructions for the critical warp hinders performance. Hence, it is important to orchestrate the fetch and issue schedulers, i.e., the fetch scheduler should prioritize fetching instructions for the critical warp rather than the next warp in the I-Buffer (in a round-robin fashion). MWF accurately identifies the critical warp, and executes critical-warp instructions from the I-Buffer. It is important that the fetch scheduler continues fetching critical-warp instructions in subsequent cycles so that the I-Buffer is continuously filled with critical-warp instructions, so that the issue scheduler in its turn can continue issuing instructions from the critical warp in subsequent cycles.

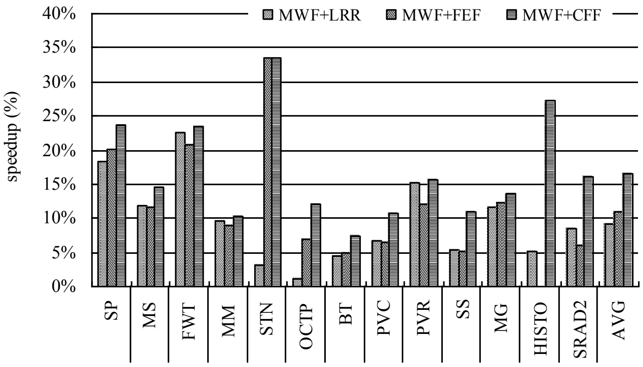

Figure 9: Performance of the CFF, FEF and LRR fetch policies, assuming the MWF issue scheduler, over baseline LRR warp scheduling.

To further analyze this critical interplay between the fetch and issue policies, and to highlight the importance of orchestrating the fetch and issue schedulers on critical warps, we now consider and compare against a fetch policy that selects the warp to fetch instructions from that has the fewest entries in the I-Buffer. We call this fetch policy *fewest-entries-first (FEF)*. In contrast to CFF, FEF is critical-warp unaware. We compare FEF against the LRR and CFF fetch policies while considering the MWF issue scheduler, see Figure 9. CFF outperforms the other fetch policies by a significant margin. Although the FEF fetch policy outperforms the LRR fetch policy on average, CFF provides a significant improvement over FEF, i.e., MWF+CFF improves performance by 17% on average over the baseline LRR issue policy, compared to 11% for MWF+FEF. This emphasizes that it is important to orchestrate fetch and issue for critical warps at barriers, and have the fetch scheduler select instructions from the critical warp in the context of barrier synchronization.

6.3 Latency Breakdown Analysis

To gain more insight as to why BAWS outperforms prior warp

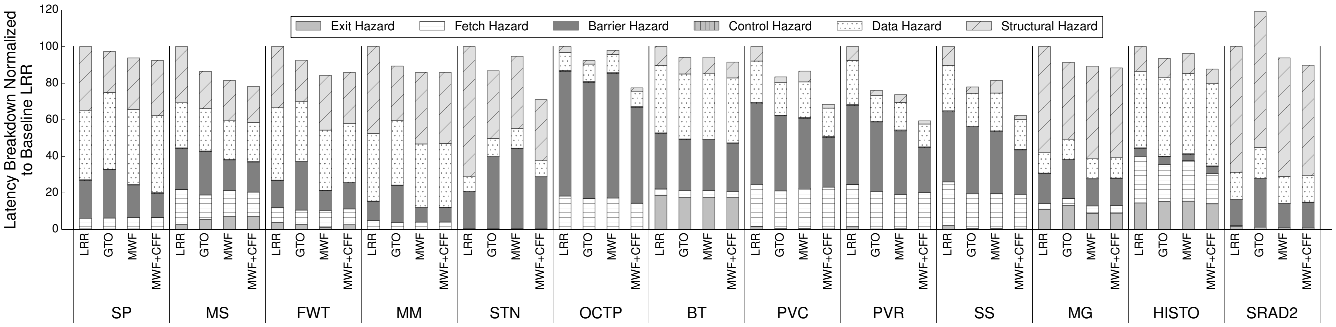

Figure 10: Latency breakdown for barrier-intensive benchmarks executed with warp scheduling algorithm LRR, GTO, MWF, and MWF(LRR) + CFF.

scheduling algorithms, we breakdown the stall cycle latency of our proposed BAWS and compare it with LRR and GTO. We identify stall cycles for control, data, structural, barrier, exit and fetch hazards. A control hazard occurs upon a taken branch or function call for a particular warp, and there no other warps to help hide the branch resolution latency. A data hazard occurs when a warp has to wait on a real data dependence to be resolved (e.g., a dependence on an older load miss). A structural hazard occurs when a decoded instruction cannot proceed execution due to the unavailability of a register file bank or functional unit. Barrier and exit hazards are a result of barrier synchronization (both explicit and implicit at the end of a TB), as extensively discussed in the paper. A fetch hazard is due to an empty I-Buffer entry as a result of a mismatch between the warp scheduling algorithms at the fetch and issue stages or due to an I-cache miss.

Figure 10 shows the stall cycle latency breakdown for LRR, GTO, MWF(LRR), and MWF(LRR)+CFF; the bars are normalized to the stall cycle breakdown for LRR. This graph reveals that most of the benefits of BAWS come from reducing the number of stall cycles due to barrier and structural hazards. Compared to LRR, MWF reduces barrier stall cycles for all benchmarks except for STN. The reason for this exception is that STN employs global synchronization across TBs which is not a built-in mechanism in GPGPUs, and which BAWS does not consider. (This is left for future work.) Combining MWF with CFF further reduces the number of stall cycles due to barriers. This is a result of removing the warp scheduling mismatch between the fetch and issue stages via CFF. Overall, BAWS (MWF+CFF) reduces more barrier hazards than GTO for all benchmarks.

Next to reducing barrier hazards, BAWS also reduces the number of stall cycles due to structural hazards. As mentioned before, warps severely contend for compute resources under LRR. In contrast, BAWS (MWF+CFF) assigns higher priority to critical warps than non-critical warps and allows the critical warps to make faster progress, thereby reducing compute resource contention. Note that GTO reduces stall cycles due to structural hazards even more than BAWS. This is because GTO greedily executes one warp as far as possible, while BAWS does not. The reduction in the number of stall cycles due to barriers however outweighs the increase in stall cycles due to structural hazards, yielding a net performance improvement for BAWS over GTO.

We also note that CFF reduces the number of fetch hazards compared to the LRR fetch policy. In other words, CFF typically reduces the number of times the warp scheduler may pick a warp that does not have any instructions in the I-Buffer, see for example MS, OCTP, SS and HISTO. The reason, as mentioned in the previous section, is that it is important to orchestrate the fetch and issue policies so that the fetch scheduler selects critical-warp instructions for the issue scheduler to execute in subsequent cycles.

Finally, BAWS also reduces the number of stall cycles due to data hazards for some benchmarks, see for example PVC, PVR and SS. Because BAWS prioritizes critical warps, and because the warps within a TB have the same priority, BAWS will schedule critical warps within a TB to run together. In case warps within the same thread block share data, i.e., there is inter-warp/intra-TB data locality, BAWS will benefit from improved data locality. GTO benefits from data locality in a very similar way, in contrast to LRR.

In summary, besides considering the barrier behavior, BAWS (MWF+CFF) combines the advantages of LRR and GTO, by reducing barrier hazards of GTO and structural/data hazards of LRR, while not reducing the structural/data hazards of LRR as much as GTO does.

6.4 Comparison Against Wider I-Buffer

CFF is conceived to bridge the mismatch between the fetch and issue stages. One may argue that widening the I-Buffer which sits between the fetch and issue stages may solve the exact the same problem. We argue that CFF incurs far less hardware overhead than widening the I-Buffer. Doubling the I-Buffer size incurs twice the storage cost in hardware; CFF on the other hand does not incur additional hardware storage cost for orchestrating the fetch and issue stages, once MWF is implemented. In addition, increasing the I-Buffer size is not as effective as CFF at widening this mismatch between fetch and issue. We conduct an experiment to evaluate the impact of I-Buffer size on performance. We double the I-Buffer size from 2 (default) to 4 instructions per warp, and we observe an average performance improvement of 0.3%, and 1.4% at most for one benchmark (MG). CFF in contrast improves performance by 1.7% on average for GTO, and 6.7 to 7.1% for MWF. We conclude that orchestrating the fetch and issue scheduling policies through CFF is both effective and efficient.

6.5 Non-Barrier-Intensive Workloads

Thus far, we focused exclusively on the evaluation for barrier-intensive workloads. However, a good barrier-aware warp scheduling policy should not hurt the performance of non-barrier-intensive workloads significantly, and if possible, it should even improve performance.

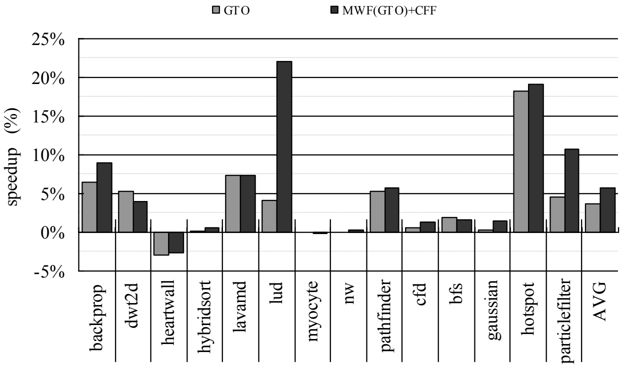

Figure 11: Performance of GTO and MWF(GTO)+CFF for non-barrier-intensive GPGPU benchmarks, normalized to LRR.

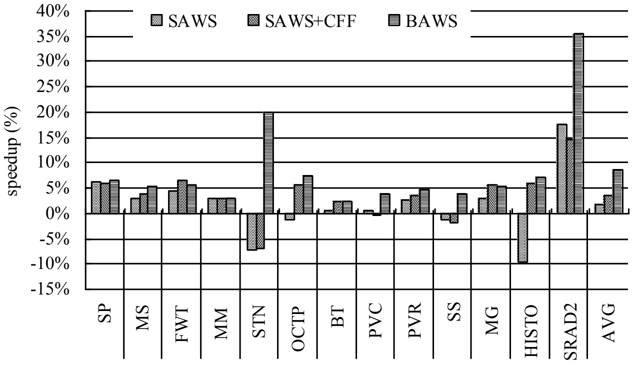

Figure 12: Relative performance for BAWS, SAWS and SAWS+CFF over GTO for the barrier-intensive benchmarks.

BAWS has minimal impact on the performance of non-barrier-intensive benchmarks because it assigns the warp scheduling priorities according to on-line detection of barriers in TBs. Thus, if there are no barriers in a kernel, BAWS degenerates into a combination of LRR or GTO (for the warps within a TB) and GTO (for selecting warps in the TB with the smallest TB-*id* between TBs). If a kernel has barriers but the performance degradation caused by the barriers themselves is small, BAWS can reduce the structural hazards significantly but improvements on the barrier hazard itself are small.

We select 14 non-barrier-intensive GPGPU benchmarks from the Rodinia benchmark suite to evaluate the impact of BAWS on non-barrier-intensive workloads, see Figure 11. BAWS (MWF+CFF) does not hurt the performance of non-barrier-intensive GPGPU kernels significantly. On the contrary, BAWS improves performance by 5.7% on average over LRR. For some benchmarks, we even observe a significant performance improvement, up to 22% for lud and 18% for hotspot. For lud, this is because it has a lot of memory accesses with good spatial locality and BAWS can leverage this locality well; for hotspot, this is because BAWS reduces structural hazards.

Although BAWS cannot improve performance much over GTO, we still observe some improvement, see for example lud, backprop and particlefilter. This is because these benchmarks do exhibit some barrier intensity, although less than the 15% threshold we used to classify benchmarks as barrier-intensive, i.e., barrier intensity equals 14%, 7% and 5% for lud, backprop and particlefilter, respectively. We observe a performance degradation of 2.6% for heartwall, but BAWS' performance is on par with GTO. BAWS performs worse than GTO for dwt2d because GTO always selects the first warp to execute first (which coincidentally turns our to be most critical warp). BAWS detects the critical warp when at least one warp has arrived at a barrier, which occurs later in the execution, yielding a small performance degradation compared to GTO.

6.6 Comparison against SAWS

Very recently published and concurrent work, called SAWS [21], also addresses GPGPU synchronization, but it focuses on the synchronization between warp schedulers, and does not target barriers within warp-phases. We conduct an experiment to compare the performance of SAWS and BAWS (i.e., MWF(GTO)+CFF) using the barrier-intensive benchmarks, see Figure 12. SAWS does improve performance over GTO for a number of benchmarks, but there are a couple benchmarks for which SAWS degrades performance, see STN and HISTO. In contrast, BAWS outperforms GTO and SAWS for all benchmarks, with an average improvement of 7% compared to SAWS and up to 27% for STN and 18% for SRAD2. Although changing the fetch policy for SAWS from LRR to CFF improves performance somewhat on average, it results in significantly less performance than BAWS.

The reason why BAWS outperforms SAWS is as follows. SAWS uses 'first-hit-time' on a barrier from a certain thread block to determine the priority between different thread blocks. As such, the thread block that hits a barrier first gets higher priority. However, that thread block may not be the one that is most critical. To make this clear, we employ an example as shown in Figure 13. At time $t0$, warps in TB0 have the highest priority, for both SAWS and BAWS. At time $t1$, TB1 will get the highest priority for BAWS because TB1 has two warps waiting at the barrier. In contrast, for SAWS, TB0 will still be considered the most critical TB because it hit its respective barrier first (namely at $t0$). At time t2, TB2 becomes the most critical TB under BAWS (because there are three warps waiting at the barrier in TB2), while TB0 remains the most critical TB under SAWS. (Note though that although a particular TB has the highest priority, this does not imply that warps from other TBs cannot make forward progress. These lower-priority warps may execute if higher-priority warps stall because of other stall reasons.) This exact scenario happens for SRAD2: the last warp in the last TB is the critical warp, however SAWS does not prioritize that warp in contrast to BAWS, which explains the 18% performance improvement of BAWS over SAWS. Another advantage of BAWS is that it balances the execution of a barrier across TBs. STN is a benchmark that has a global barrier as a synchronization point across TBs. BAWS better balances the execution across TBs towards this barrier, which explains the 27% performance benefit of BAWS over SAWS.

We further conduct experiments to evaluate the effect of BAWS and SAWS on the non-barrier-intenisve benchmarks, see Figure 14. SAWS and BAWS are performance-neutral compared to GTO for all the non-barrier-intensive benchmarks, with lud being the most notable outlier. (Note that the fraction of stall cycles caused by barriers equals 14.8%, slightly below our cutoff at 15% to be considered barrier-intensive.) The reason why both SAWS and BAWS significantly improve performance over GTO is that lud contains a lot of memory accesses which exhibits good spatial locality within a TB. BAWS and SAWS can better leverage this intra-TB locality than GTO which can only

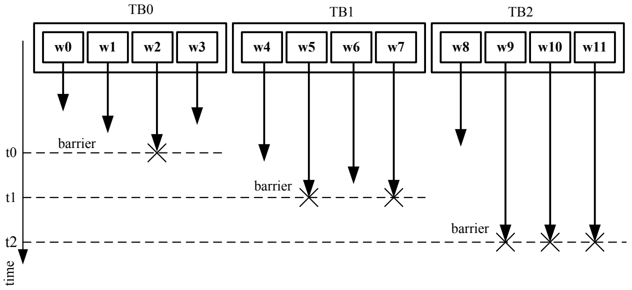

Figure 13: Warp scheduling: BAWS versus SAWS.

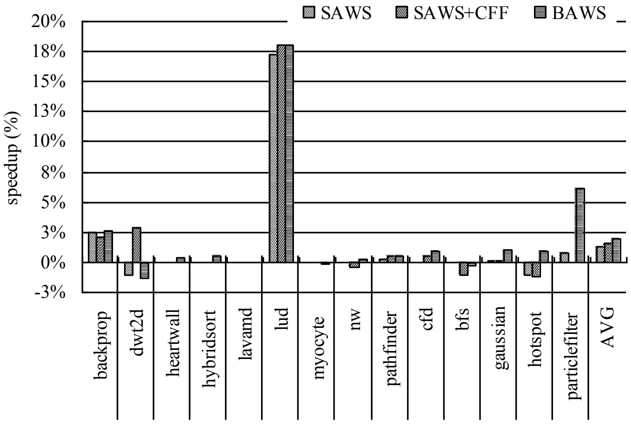

Figure 14: Relative performance for BAWS, SAWS and SAWS+CFF over GTO for the non-barrier-intensive benchmarks.

exploit intra-warp locality. This explains the performance benefit of 17% and 18% for SAWS and BAWS over GTO, respectively.

7. RELATED WORK

In this section, we first describe related work on warp scheduling policies. We subsequently compare against prior studies on GPGPU synchronization.

7.1 Warp Scheduling

Most studies on warp scheduling focus on reducing the performance loss caused by long latencies to memory. Jog et al. [14] propose a prefetch-aware warp scheduling policy. Gebhart et al. [8] propose a two-level thread scheduler to hide both local memory and global memory latencies. To reduce the performance degradation caused by long-latency operations and conditional branch instructions, Narasiman et at. [23] propose a two-level warp scheduling algorithm and a large warp microarchitecture. Rogers et al. [26] study the cache behavior of GPGPU workloads and they propose a set of cache-aware warp scheduling policies such as GTRR, GTO, and CCWS, which are effective for cache-sensitive GPGPU workloads. Several other memory-related warp scheduling algorithms have been proposed, see for example [12, 13, 16, 20, 22, 28]. Concurrent work by Park et al. [25] maximizes memory-level parallelism for GPUs and also coordinates warp and fetch scheduling, as we propose in the context of barriers with CFF.

A second flavor of warp scheduling algorithms focus on how to improve the resource utilization of GPGPUs, as due to branch divergence, the SIMD lanes are often underutilized. Fung et al. [22] propose a hardware approach to regroup threads into new warps on the fly following the occurrence of diverging branch outcomes. Rogers et al. [27] propose a divergence-aware warp scheduler to improve the performance and energy of divergent applications. Recently, Lee et al. [17] propose a TB scheduler as well as a combined TB-plus-warp scheduler to improve overall resource utilization.

Some prior work addresses warp-level-divergence. Xiang et al. [31] propose to allocate and release resources at the warp level rather than at the TB level. As a result, the number of active warps is increased while not increasing the size of the critical resources. Lee et al. [19] also observe significant execution disparity for warps within a TB. They define the slowest warp in a TB as the critical warp and they propose a criticality-aware warp scheduling (CAWS) policy driven by application program hints to improve the performance of GPGPU applications. These works address execution disparity at the exit points of a thread block, which we find to be a minor contributor to the total time stalled on barriers in barrier-intensive workloads, see Figure 1. BAWS, in contrast, tackles execution disparity at the warp-phase level. Follow-on work by Lee et al. [18] proposes CAWA which predicts warp criticality based on dynamic instruction count and the number of stall cycles, to account for warp imbalance and shared resource contention. CAWA steers scheduling and cache partitioning by warp criticality. CAWA does not take barrier synchronization into account as we do in BAWS.

Very recent and concurrent work by Liu et al. [21], called SAWS, optimizes synchronization performance between warp schedulers, but does not target barriers within warp-phases. Furthermore, SAWS prioritizes the first TB hitting a barrier, which may not be the most critical TB. BAWS on the other hand optimizes synchronization performance within warp-phases while optimizing the most critical TB, yielding a substantial performance benefit over SAWS as demonstrated in the previous section.

In summary, the above warp scheduling polices improve the performance of GPGPU applications from different aspects, such as long memory latency, branch divergence, and thread block exit points. However, none of these characterize and address barrier behavior within a thread block. In contrast, our work comprehensively characterizes the barrier behavior of barrier-intensive GPGPU applications, and proposes a barrier-aware warp scheduling (BAWS) policy to reduce fine-grained barrier-induced warp execution disparities.

7.2 GPGPU Synchronization

GPGPUs provide hardware supported synchronization within a TB but not between TBs. Whereas this paper focuses on synchronization within a TB, prior work addressed synchronization issues between TBs. Xiao and Feng propose three techniques — simple synchronization, tree-based synchronization, and lock-free synchronization — to achieve fast synchronization between TBs [7, 32]. Yilmazer et al. [33] propose a hardware-based blocking synchronization mechanism that uses hierarchical queuing for scalability and efficiency for synchronization-intensive GPGPU applications. To optimize thread-level parallelism for GPGPUs, Kayiran et al. [15] propose a dynamic TB scheduling mechanism. They model the synchronization between TBs by using atomic instructions, but they do not come up with an approach to improve synchronization performance. For the heterogeneous architectures consisting of GPGPUs and CPUs, Guo et al. [9, 10] propose a code generator with three features of which one is an instance-level instruction scheduler for synchronization relaxation. All of these GPGPU synchronization studies focus beyond a TB, while our work studies synchronization behavior within a TB.

8. CONCLUSION

In this paper, we observe that barrier-intensive GPGPU applications can be stalled significantly on barriers. In general, these stall cycles are caused by execution divergence of warps in a warp-phase in a TB, and we define such divergence as warp-phase-divergence. We identify a number of causes of warp-phase-divergence: application code, input data, shared resource contention, and warp scheduling algorithms.

To mitigate barrier induced stall cycle inefficiency, we propose barrier-aware warp scheduling (BAWS), a novel warp scheduling policy that combines two different techniques: most-waiting-first (MWF) warp scheduling for the issue stage, and critical-fetch-first (CFF) warp scheduling for the fetch stage. MWF assigns a higher scheduling priority to warps of a thread block that has a larger number of warps waiting at a barrier. CFF orchestrates the fetch and issue schedulers, and fetches instructions from the warp to be issued by MWF in the next cycle. We evaluate BAWS (MWF+CFF) and compare it against LRR and GTO for a set of 13 barrier-intensive benchmarks. The experimental results show that BAWS speeds up the barrier-intensive benchmarks over LRR by 17% on average (and up to 35%), and by 9% on average (and up to 30%) over GTO. BAWS outperforms SAWS by 7% on average and up to 27%. Moreover, for non-barrier-intensive workloads, we report that BAWS is performance-neutral compared to GTO and SAWS, while improving performance by 4% on average (and up to 17%) over LRR. Hardware cost for BAWS is as small as 6 bytes per SM.

Acknowledgements

We thank the reviewers for their thoughtful comments and suggestions. This work is supported by the China 973 Program under No. 2015CB352400, the major scientific and technological project of Guangdong province (2014B010115003), Shenzhen Peacock Innovation project (KQCX20140521115045448), outstanding technical talent program of CAS; NSFC under Grant No. 61232008, 61272158, 61328201, U1401258, and 61472008; the 863 Program of China under Grant No. 2012AA010905 and 2015AA015305. Lieven Eeckhout is partly supported by a Chinese Academy of Sciences (CAS) visiting professorship for senior international scientists, and through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement No. 259295.

9. REFERENCES

[1] CUDA programming guide, version 3.0. *NVIDIA CORPORATION*, 2010.

[2] ATI stream technology. *Advanced Micro Devices,Inc. http://www.amd.com/stream.*, 2011.

[3] OpenCL. *Khronos Group. http://www.khronos.org/opencl.*, 2012.

[4] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, 2009.

[5] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 57–64, 2008.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[7] Wu-Chun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pages 3801–3804, 2010.

[8] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 235–246, 2011.

[9] Ziyu Guo, Bo Wu, and Xipeng Shen. One stone two birds: Synchronization relaxation and redundancy removal in GPU-CPU translation. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 25–36, 2012.

[10] Ziyu Guo, Eddy Zheng Zhang, and Xipeng Shen. Correctly treating synchronizations in compiling fine-grained SPMD-threaded programs for CPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 310–319, 2011.

[11] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: A MapReduce framework on graphics processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 260–269, 2008.

[12] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283, 2014.

[13] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 395–406, 2013.

[14] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Orchestrated scheduling and prefetching for GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 332–343, 2013.

[15] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 157–166, 2013.

[16] Nagesh B Lakshminarayana and Hyesoon Kim. Effect of instruction fetch and memory scheduling on GPU performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[17] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the International*

Symposium on High Performance Computer Architecture (HPCA), pages 260–271, 2014.

[18] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. CAWA: Coordinated warp scheduling and cache priorization for critical warp acceleration of GPGPU workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 515–527, 2015.

[19] Shin-Ying Lee and Carole-Jean Wu. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 175–186, 2014.

[20] Dong Li, Minsoo Rhu, Daniel R Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S Fussell, and Stephen W Keckler. Priority-based cache allocation in throughput processors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2015.

[21] Jiwei Liu, Jun Yang, and Rami Melhem. SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 383–394, 2015.

[22] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 235–246, 2010.

[23] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 308–317, 2011.

[24] CUDA Nvidia. CUDA SDK code samples.

[25] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. ELF: Maximizing memory-level parallelism for gpus with coordinated warp and fetch scheduling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page article 8, 2015.

[26] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the IEEE International Symposium on Microarchitecture (MICRO)*, pages 72–83, 2012.

[27] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Divergence-aware warp scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 99–110, 2013.

[28] Ankit Sethia, D Anoushe Jamshidi, and Scott Mahlke. Mascar: Speeding up GPU warps by reducing memory pitstops. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185, 2015.

[29] Inderpreet Singh, Arrvindh Shriraman, Wilson Fung, Mike O'Connor, and Tor Aamodt. Cache coherence for GPU architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590, 2013.

[30] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign (UIUC), 2012.

[31] Ping Xiang, Yi Yang, and Huiyang Zhou. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 284–295, 2014.

[32] Shucai Xiao and Wu-Chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010.

[33] Ayse Yilmazer and David Kaeli. HQL: A scalable synchronization mechanism for GPUs. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 475–486, 2013.