iPAWS : Instruction-Issue Pattern-based Adaptive Warp Scheduling for GPGPUs

Minseok Lee, Gwangsun Kim, John Kim
KAIST
Daejeon, Korea
{lms135, gskim, jjk12}@kaist.edu

Woong Seo, Yeongon Cho, Soojung Ryu
Samsung Electronics
Giheung, Korea
{brand.seo, yeongon.cho, soojung.ryu}@samsung.com

ABSTRACT

Thread or warp scheduling in GPGPUs has been shown to have a significant impact on overall performance. Recently proposed warp schedulers have been based on a *greedy* warp scheduler where some warps are prioritized over other warps. However, a single warp scheduling policy does not necessarily provide good performance across all types of workloads; in particular, we show that greedy warp schedulers are not necessarily optimal for workloads with *inter*-warp locality while a simple round-robin warp scheduler provides better performance. Thus, we argue that instead of single, static warp scheduling, an *adaptive* warp scheduler that dynamically changes the warp scheduler based on the workload characteristics should be leveraged. In this work, we propose an *instruction-issue pattern-based adaptive warp scheduler* (iPAWS) that dynamically adapts between a greedy warp scheduler and a fair, round-robin scheduler. We exploit the observation that workloads that favor a greedy warp scheduler will have an instruction-issue pattern that is biased towards some warps while workloads that favor a fair, round-robin warp scheduler will tend to issue instructions across all of the warps. Our evaluations show that iPAWS is able to adapt to the more optimal warp scheduler dynamically and achieve performance that is within a few percent of the statically determined, more optimal warp scheduler. We also show that iPAWS can be extended to other warp schedulers, including the cache-conscious wavefront scheduling (CCWS) and Memory Aware Scheduling and Cache Access Re-execution (MASCAR) to exploit the benefits of other warp schedulers while still providing adaptivity in warp scheduling.

1. INTRODUCTION

Manycore accelerators such as GPGPUs with its significant computing capability are becoming widely utilized for different workloads [5, 24, 16]. With programming models such as CUDA [18] or OpenCL [11], these accelerators enable thousands of threads to be executed in parallel. The GPGPU architectures have a hierarchy of threads – a collection of threads form a warp (or a wavefront) and a group of warps form a CTA (cooperative thread array) or a thread block. Within the GPGPU architecture, a scheduler needs to determine which group of threads (or which warp) should be executed by the core. This scheduler is referred to as the warp or wavefront scheduler and there has been significant amount of research [20, 21, 12, 15, 8, 9, 6, 22] done on different warp schedulers to improve overall performance. Different warp schedulers that have been proposed exploit different aspects of GPGPU architectures – including increasing cache locality, overcoming branch divergence, and hiding long memory latencies. These prior work have shown performance improvements on the target workloads which exhibit such problems while having minimal impact on compute-intensive workloads. However, a single warp scheduler does not necessarily provide the best performance across all workloads. As a result, we propose to adopt a classical approach in computer architecture – an *adaptive* warp schedulers that adapts between different warp schedulers based on the workload characteristics.

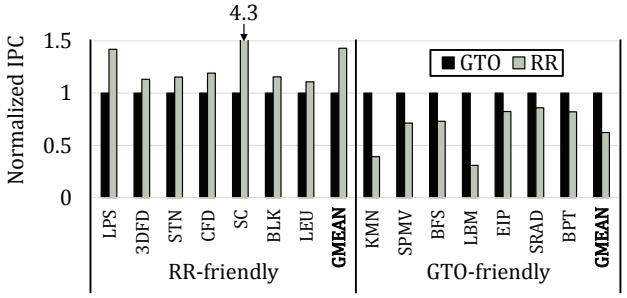

Figure 1: Performance comparison of a greedy scheduler (GTO) and a fair, round-robin (RR) scheduler across different GPGPU workloads.

Most of the recently proposed warp schedulers have been based on a greedy scheduler [20] that favors or prioritizes some (or some set) of warps over the remaining warps. However, in this work, we first show that such greedy approach is not necessarily optimal across all workloads. Figure 1 shows a comparison of two common, baseline warp schedulers – a fair, round-robin (RR) warp scheduler and a greedy warp scheduler (greedy-then-oldest (GTO) [20]). The simulation setup is described later in Section 2.1 and the results are normalized to the performance of GTO. As shown in prior work [20], a greedy warp scheduler where older warps are continually prioritized over younger warps better exploits *intra*-warp locality. Thus, for workloads with intra-warp locality (e.g., `KMN`, `LBM`), GTO exceeds the performance of RR scheduler, by up to 3.2×, and on average, 60% across the different workloads. However, some workloads have limited

978-1-4673-9211-2/16/$31.00

intra-warp locality but have *inter*-warp locality – thus, continuously prioritizing older warps can not properly exploit this inter-warp locality. For example, SC workload has significant inter-warp locality and the performance of RR exceeds the GTO by 4.3×. [1] While the L1 hit rate is approximately 38% with RR scheduling, the hit rate drops to 9% with GTO scheduling. On average, RR exceeds the performance of GTO by 43% across workloads where a greedy approach does not necessarily provide the best performance. In this work, we refer to workloads where the performance of RR exceed the performance of GTO as *RR-friendly* workloads while the workloads where GTO exceeds the performance of RR are referred to as *GTO-friendly* workloads.

Since a single scheduler does not necessarily provide the best performance across all workloads, we propose an *adaptive* warp scheduler referred to as Instruction-Issue Pattern-based Adaptive Warp Scheduling (iPAWS). iPAWS adapts between a fair, round-robin (RR) warp scheduler and a greedy warp scheduler during runtime based on the instruction-issue pattern of the warps. We exploit the observation that for workloads that prefer a greedy warp scheduler, instruction-issue pattern will be biased towards the older warps while for workloads that prefer a fair, RR warp scheduler, instruction-issue pattern will tend to be equally distributed across the warps. The proposed iPAWS adapts between two simple RR and GTO warp schedulers, but iPAWS can be extended to other warp schedulers since many recent warp schedulers are based on a greedy warp scheduler. We show how iPAWS can be extended to adapt between RR and cache-conscious wavefront scheduling (CCWS) [20] or adapt between RR and Memory Aware Scheduling and Cache Access Re-execution (MASCAR) [22]. In addition, we also show how iPAWS can be extended for concurrent kernel execution with multiple kernels executing in each core.

In particular, the contributions of this paper include the following.

- We show that a single warp scheduler is not necessarily optimal across different workloads. In particular, we show that previously proposed greedy warp-schedulers are not optimal for workloads with *inter*-warp locality and a fair, round-robin scheduler outperforms a greedy warp scheduler.
- We propose Instruction-Issue Pattern-based Adaptive Warp Scheduling (iPAWS) that adapts between a greedy, warp scheduler and a fair, round-robin warp scheduler based on the instruction-issue pattern across the different warps within a core.
- We extend iPAWS to adapt between other recently proposed warp schedulers, including RR and CCWS [20] as well as RR and MASCAR [22]. In addition, we show how iPAWS can be extended to concurrent kernel execution.
- Our evaluation shows that our proposed adaptive scheduler can adapt to the more optimal warp scheduler and achieve performance that is within a few percent of the more optimal scheduler that is statically determined.

[1]The results for SC are not consistent with prior work [20] where GTO had slightly *better* performance compared to RR. As we discuss in Section 5, the workload inputs and parameters can impact the performance and the effect of the warp scheduler. The results shown in Figure 1 used an input dimension parameter of 256 [5] while prior work [20] could possibly have used a smaller input parameter.

Table 1: Baseline Configuration

Parameter	Value
# of cores	15
Core	MAX 1536 threads, Warp size: 32, 32768 registers, 1400 MHz 48KB shared memory
Constant cache	8KB
Texture cache	32KB, 16-way, 64B line
L1 Data cache	32KB, 8-way, 128B line
L2 Cache	128KB/MC, 8-way, 128B line
DRAM model	FR-FCFS, 6MCs, 16 banks/MC, 924 MHz
DRAM timing	t_{CL}=12, t_{RP}=12, t_{RC}=40 t_{RAS}=28, t_{RCD}=12, t_{RRD}=6
Network topology	crossbar
Network clock	1400 MHz

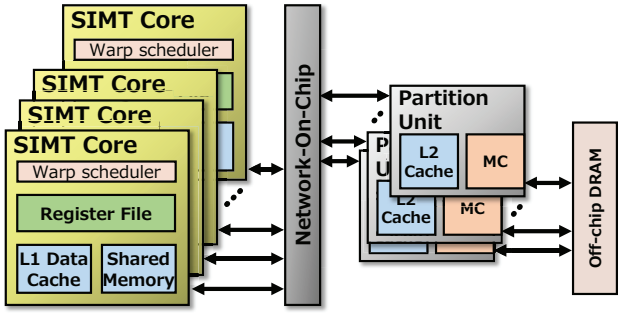

Figure 2: Block diagram of the baseline architecture.

2. WORKLOAD ANALYSIS

In this section, we first describe our evaluation methodology and provide an overview of the GPGPU architecture that we consider. We then provide an analysis of the different workloads and how different warp schedulers impact their performance.

2.1 Methodology

We use a detailed GPGPU simulator (GPGPU-sim v3.2) [4] in our evaluation and our configuration parameters are described in Table 1. The GPGPU that we model consists of 15 cores connected to 6 memory controllers, as shown in Figure 2 and the parameters are roughly based on the GTX480. The simulator was modified to implement the adaptive warp scheduler that we propose in this work. Our baseline GTO scheduler is based on CCWS implementation that is publicly available [20]. We considered a wide range of GPGPU CUDA workloads, including applications from Rodinia [5], Parboil [24], NVIDIA SDK [16], and workloads from GPGPU-sim [4], as summarized in Table 2. In addition to RR-friendly (RF) workloads and GTO-friendly (GF) workloads, we also evaluate various scheduler-neutral (NEU) workloads where the warp scheduler has minimal impact on overall performance. For all the workloads, the simulation is done until the completion of the workloads.

Table 2: GPGPU Workload Description

Name	Abbr.	Type
Kmeans[5]	KMN	GF
Sparse Matrix-Dense Vector Multiplication[24]	SPMV	GF
Lattrice-Boltzmann Method[24]	LBM	GF
Breadth First Search[5]	BFS	GF
B+Tree[5]	BPT	GF
EstimatePInlineP[16]	EIP	GF
Structured Grid[5]	SRAD	GF
3D Laplace Solver[4]	LPS	RF
3D finite difference[16]	3DFD	RF
7-point stencil-CFG2[24]	STN	RF
Black-Schole option pricing[16]	BLK	RF
CFD Solver[5]	CFD	RF
Leukocyte[5]	LEU	RF
Streamcluster[5]	SC	RF
VectorAdd[16]	VADD	NEU
Back Propagation[5]	BKP	NEU
Matrix Multiplication[16]	MMUL	NEU
PathFinder[5]	PDFR	NEU
Ray Tracing[4]	RAY	NEU
ScanLargeArray[16]	SCAN	NEU
MUMmerGPU[4]	MUM	NEU
Fast Fourier transform [24]	FFT	NEU
Sum of Absolute Differences[24]	SAD	NEU
BinomialOptions[16]	BNEUO	NEU

2.2 Workload Characteristics

To understand the workload characteristics, we first analyze the impact of different warp schedulers on cache locality. Figure 4(a) plots the different cache locality for GTO-friendly workloads and the statistics are generated similar to [20] but extended to measure shared L2 cache locality. All of the workloads show significant intra-warp locality and the number of L1 data cache hits from intra-warp locality (L1 Intra-Hit) is increased when using GTO scheduler compared to RR. For SPMV, it shows not only intra-warp locality but also shows considerable inter-warp locality in L1 cache (L1 Inter-Hit). However, the number of hits by inter-warp locality in L1 cache is similar for both schedulers because inter-warp locality in this specific workload targets small data set. However, for two of the workloads (BPT and SRAD), there is minimal difference in the locality between GTO and RR. Even though these two workloads do not have significant intra-warp locality, the greedy scheduler still outperforms RR scheduler by approximately 20% (Figure 1). The difference for these workloads occurs from the DRAM contention and memory access pattern (Figure 5). For both BPT and SRAD, GTO improves the AMAT (Average Memory Access Time) by reducing the contention at the different banks while RR reduces the bank-level parallelism and thus, increases AMAT. As a result, simply only using locality information is insufficient to properly adapt the warp scheduler.

In comparison to intra-warp locality, inter-warp locality can be exploited when warps progress at a similar rate. Figure 3 shows a simplified pseudo-code for a stencil workload (`LPS`) where inter-warp locality can be exploited. Each thread does a stencil operation with the `main_data` and the `halo_data` that is loaded in line 4 and line 8. After completing the stencil operation, each thread moves on to the next tile by incrementing the index appropriately in line 13 and 14. While each thread does not reuse the data that it loads into the L1/L2cache, the data can be reused by other threads (or warps) – e.g., the halo_data can be used by a neighboring thread as its main_data, etc – if the threads progress at a similar rate. Thus, a fair, round-robin warp scheduler can better exploit such inter-warp locality.

```
   //Simplified example with inter-warp locality
1: for(int i=0; i<dimz; i++) {
2:     ...
3:     if(active) {
4:         main_data = g_input[global_idx];
5:     }
6:     ...
7:     if(halo) {
8:         halo_data = g_input[global_halo_idx];
9:     }
10:    ...
11:    global_idx += stride;
12:    global_halo_idx += stride;
13:    ...
14: }
```

Figure 3: Example pseudo-code from `LPS` [4] workload with inter-warp locality.

Figure 4(b) shows cache locality for RR-friendly workloads. If GTO is used instead of RR, the number of hits from inter-warp locality in L1/L2 cache is decreased. For example, for LPS, the inter-*miss* increases by 22% when GTO scheduler is used instead of RR scheduling and by 21% for SC workload. For CFD, by using GTO, inter-warp locality is lost in L1 cache and it results in hits in L2 cache. Even though BLK was classified as a RR-friendly workload, it does not have any cache locality. However, the inter-warp locality from BLK is exploited in the row-buffer of the main memory instead of the L1/L2 cache (Figure 5). By using a greedy (GTO) warp scheduler, some warps outpace other warps and thus, the row buffer locality can not be fully exploited while RR scheduler helps to preserve the row-buffer locality. LEU is another workload where there is minimal difference in the cache locality but there is a load imbalance across the warps such that RR scheduler outperforms GTO by more than 10%.

In summary, while the impact of the more optimal warp scheduler can be characterized by the amount of cache locality for many workloads, other workloads are impacted by the DRAM access pattern or other resource utilization. As a result, instead of leveraging a particular metric related to cache locality or memory access pattern, we propose an adaptive scheduler that exploits the instruction-issue pattern across the different warps to adapt to the more optimal warp scheduler. In addition, a single set of workload inputs are commonly used to evaluate the performance of GPGPUs. However, we also show later in Section 5.2, that different workload inputs can change the runtime characteristics such as exploited cache locality and also impact the performance of warp scheduler. Thus, instead of leveraging a compiler-based approach, we propose a hardware-based adaptive warp scheduler that adapts between warp schedulers during run-time.

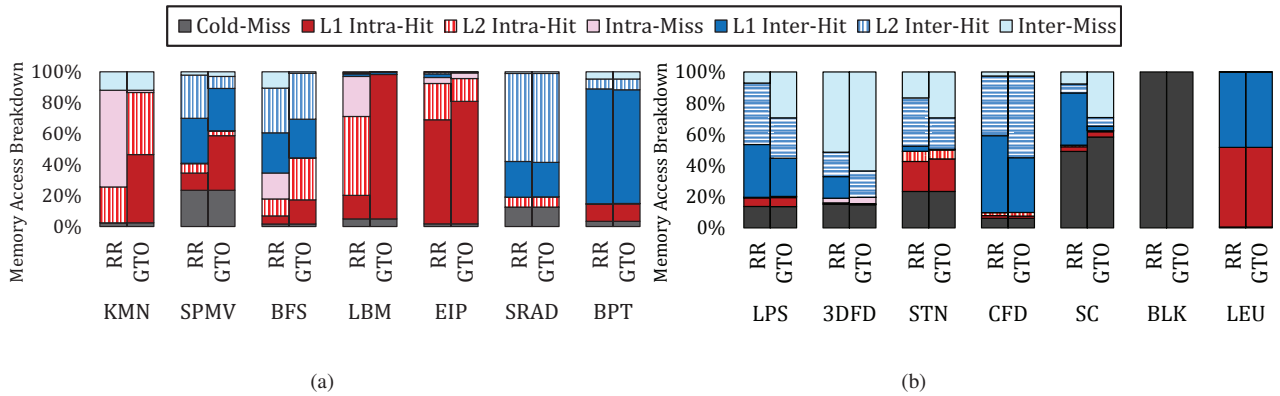

Figure 4: Cache access breakdown for (a) GTO-friendly workloads and (b) RR-friendly workloads.

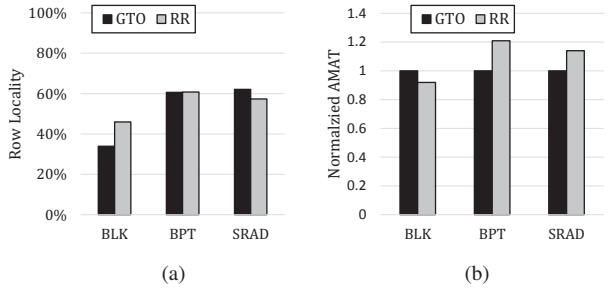

Figure 5: (a) DRAM row-buffer locality and (b) Average memory latency for workloads whose performance is not determined by cache locality but DRAM access pattern.

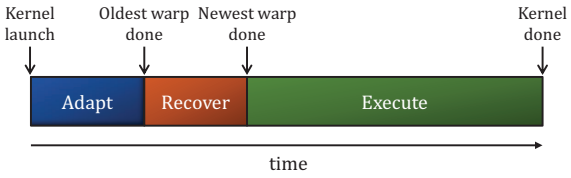

Figure 6: Three different phases of instruction-issue pattern-based adaptive warp scheduling.

3. INSTRUCTION-ISSUE PATTERN BASED ADAPTIVE SCHEDULER

In this section, we propose to leverage the instruction-issue pattern across the different warps to adaptively switch between a greedy warp scheduler and a fair warp scheduler. For a workload which favors a greedy scheduler (i.e., GTO), some of the warps will continue to be executed and favored because of the workload characteristics. However, for workloads that do not favor a greedy scheduler, even if a greedy warp scheduler is used, the instructions from different warps will likely be issued. We exploit this observation to understand the instruction-issue pattern of the workload to adaptively select the warp scheduler which provides higher performance. While our adaptive scheduler is based on switching between two, existing simple warp schedulers, the proposed adaptive scheduler is not limited to them and alternative schedulers can be used in the adaptive decision, as we discuss in Section 4.2.

3.1 Overview

A high-level overview of the proposed adaptive scheduler is shown in Figure 6 which consists three phases for each kernel. The initial `Adapt` phase determines the instruction-issue pattern and decides whether this workload is more suitable with a RR scheduler or a greedy scheduler. To determine whether the greedy approach or the fair warp scheduling provides better performance, we analyze the instruction-issue pattern or the number of instructions issued across the different warps within a core. The number of instructions issued for each warp is counted until the first warp finishes execution. Examples of different instruction-issue patterns are shown in Figure 7 where the *y*-axis represents the different warps allocated to a core and the *x*-axis is the number of instructions issued per warp. The warps shown on the bottom of *y*-axis represent older warps or the warps that will be prioritized with a greedy scheduler. The instruction-issue pattern can follow either a *concave* pattern where the curve bends *inwards* towards the two axes or a *convex* pattern where the curve bends *outwards* and away from the two axes.

In our proposed adaptive scheduler, a greedy scheduler is assumed to be the initial warp scheduler to help better identify the instruction-issue pattern of a given workload. If a greedy warp scheduler has better performance than RR scheduler, the instruction-issue pattern across the different warps will show a "concave" pattern as shown in Figure 7(a) as a few of the older warps will issue instructions more heavily compared with the other warps. For example, for workloads with intra-warp locality, warps will likely hit in the local L1 cache issue more instructions. In comparison, for workloads that perform better with a fair warp scheduler, the warp instruction-issue pattern will result in a "convex" shape as shown in Figure 7(b). For workloads with limited intra-warp locality but higher *inter*-warp locality, continuously favoring certain number of warps will result in local L1 misses – and thus, instructions from other warps will be issued. As a result, instructions will be more evenly distributed across the different warps and result in a convex shape. For workloads with convex instruction-issue pattern, we introduce a `Recover` phase to improve the performance of our adaptive scheduler (Section 3.4). Afterwards, the `Execute` phase continues with the adaptively selected warp scheduler. In the following section, we describe how we evaluate whether the instruction-issue

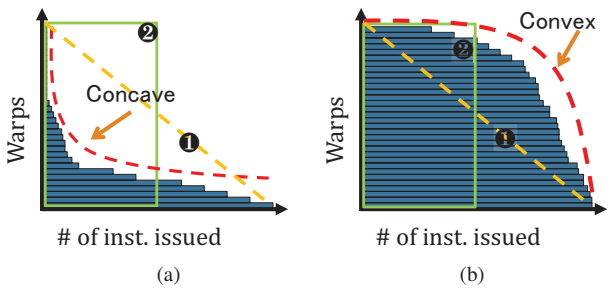

Figure 7: (a) Concave and (b) convex instruction-issue patterns.

Algorithm 1 Determining Instruction-Issue Pattern.

W = number of warps-of-interest (WOI)
$inst_i$ = number of instructions issued for warp i
$btime_i$ = barrier waiting time for warp i `//section 3.3`
`//if there is no barrier, iscore == inst`
$iscore_i = inst_i + btime_i$
$iscore_{thr} = W \times (\frac{iscore_{max}}{2})$
$iscore_{sum} = \sum_{i=0}^{W} (iscore_i)$
if $iscore_{sum} < iscore_{thr}$ **then**
  concave; `//stay with greedy scheduler`
else
  convex; `// switch to fair scheduler`
end if

pattern follows a concave or a convex pattern in the `Adapt` phase.

3.2 Implementation

A high-level description of the algorithm to determine the instruction-issue pattern is described in Algorithm 1. The evaluation to determine the instruction-issue pattern is done after the first warp (most likely the oldest warp) has completed execution. After the first warp has finished execution, the sum of *all* the instructions ($inst_{sum}$) executed in the core is compared against a threshold instruction count ($inst_{thr}$) to determine the instruction-issue pattern. The threshold instruction count ($inst_{thr}$) is essentially the triangle area (❶) shown in the Figure 7.

$$inst_{thr} = max\_num\_warp \times (\frac{inst_{max}}{2})$$

This area is also equal to the rectangle area (❷) shown in Figure 7 – which represents the number of instructions that could have been issued under a *strict* RR warp scheduler. If $inst_{sum}$ is smaller than $inst_{thr}$, the number of instructions issued is biased toward older warps and thus a greedy scheduler is more appropriate to provide higher performance. In comparison, if $inst_{sum}$ is greater than $inst_{thr}$, the workload is assumed to be have issued instructions across all of the warps and thus, a fair, RR scheduler is more appropriate. In this analysis, all of the warps allocated to the same core (including warps across the different thread blocks) are considered in the comparison.

The two examples shown earlier in Figure 7 are relatively easy to determine their instruction-issue pattern. However, if all of the warps allocated to the core do not issue any instruction during the `Adapt` phase, the instruction-issue pattern can

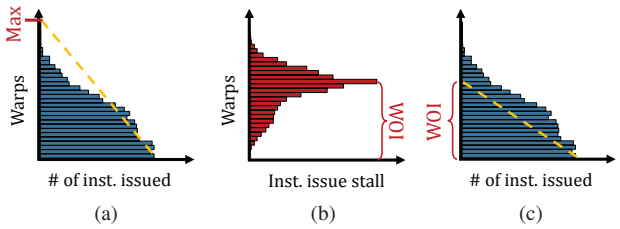

Figure 8: (a) Incorrectly determined Warps-of-Interest (WOI), (b) instruction-issue stall count for each warp and (c) correctly determined WOI for BLK workload.

be incorrectly interpreted. For example, for the instruction-issue pattern shown in Figure 8(a), if all of the warps allocated are used to determine the $inst_{thr}$ (as shown in in Figure 8(a)), the workload can be incorrectly analyzed since the workload would be determined as "concave" instruction-issue pattern. However, this workload (`BLK`) is actually RR-friendly and should be adaptively determined as a convex instruction-issue pattern. The key problem with the current approach is that we considered all of the warps allocated to the core and it is clear that some of the warps have not issued any instructions. Even if the older warps are stalled and instructions cannot be issued from these warps, it is possible that none of the newer warps can be issued and result in the core being stalled.

In order to properly determine the instruction-issue pattern of the workloads, we propose to identify the *warps-of-interest* (WOI) to determine if the instruction-issue pattern is concave or convex and adaptively determine the appropriate warp scheduler. Since not all warps actively issue instructions during the `Adapt` phase, the WOI is the appropriate number of warps to be considered in the instruction-issue pattern determination. As discussed earlier, one simple approach to determine WOI is to consider all warps allocated to the core as shown in Figure 8(a). However, this does not always accurately reflect the instruction-issue behavior. For example, with a greedy (GTO) scheduler, there can be a long "tail" – i.e., a few of the newer warps can have only a few instructions scheduled and this can also incorrectly represent the instruction-issue patterns by increasing WOI and thus, the $inst_{thr}$ metric.

To properly determine WOI, we measure the instruction-issue *stall* for each warp. For each warp, a separate counter is used to measure the number of times each warp is stalled even though it is the oldest warp. The oldest warp with the highest value of the instruction-issue stall counter value is determined as the limit for WOI – thus, any warp older than this particular warp is considered to be a part of WOI while newer warps are not considered. The intuition is that newer warps do not receive a fair opportunity to issue instructions and thus, should not be considered in determining the instruction-issue pattern. The instruction-issue stall pattern is shown in Figure 8(c) and based on the maximum value, the WOI is determined and results in a convex pattern.

3.3 Barrier Synchronization

If barrier synchronization is used within the workload, using instruction-issue pattern in the adaptive scheduler can be problematic. In CUDA workloads, barriers are used to synchronize all of the warps within a thread block as all warps

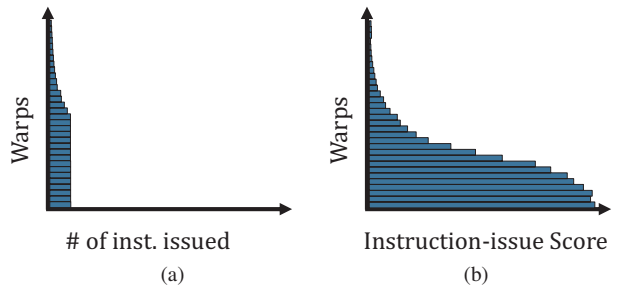

Figure 9: (a) Impact of barrier synchronization on the instruction-issue pattern for PFT workload and (b) instruction-issue pattern when barrier stalls are properly accounted for and the instruction-issue *score* is plotted. The *x*-axis has the same scale and thus (b) shows larger values with the barrier stalls cycle count added.

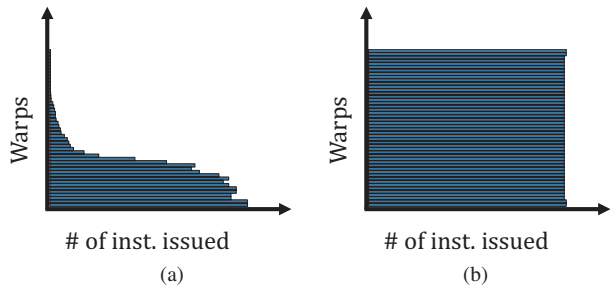

Figure 10: The instruction issue pattern for SC workload (a) after adapt phase and (b) after recovery phase.

within a thread block wait until the other warps reach the barrier [18]. As a result, this can have a significant impact on the instruction-issue pattern since all of the warps within the particular thread block will have the similar number of instructions issued. The instruction-issue pattern will resemble a convex pattern and for workloads that favor RR scheduler, this is not an issue; however, for workloads that favor GTO scheduler, this can result in an incorrect warp scheduler being adaptively selected. The problem with a barrier is more problematic for workloads where a larger number of warps exist within a thread block and the number of thread blocks allocated to each core is relatively limited.

An example of this problem is shown with an instruction-issue pattern for the PFT workload in Figure 9(a). Each thread block in PFT contains 16 warps and only two thread blocks are allocated to each core. This particular workload has intra-warp locality and a greedy warp scheduler should be adaptively selected. However, because of the barrier, the instruction-issue pattern will be identified as a convex pattern (Figure 9(a)) and RR warp scheduler will be incorrectly selected.

To address this problem associated with barriers, we introduce instruction-issue *score* where we add the number of cycles that each warp waits at a barrier (btime_i) to the number of instruction issued for that warp (inst_i). Thus, the instruction-issue score is increased when an instruction is issued or when the warp is stalled at a barrier and another warp issues an instruction. Figure 9(b) shows the updated instruction-issue pattern when the instruction-issue *score* is used. Since a greedy-warp scheduler is used as the initial scheduler, the waiting time for the older warps can be significantly longer. This results in high instruction-issue scores for the older warps and a concave instruction-issue pattern is selected.

3.4 Recover Phase

As mentioned earlier, for workloads with concave instruction-issue patterns where an adaptive decision selects the greedy warp scheduler, the warp scheduling continues with the greedy scheduler and a Recover phase is not necessary. However, for workloads with convex instruction-issue patterns where the adaptive scheduler selects a fair warp scheduler, the benefits of a fair warp scheduler might not be realized if the workload switches immediately from a greedy scheduler to a fair scheduler. For example, *inter*-warp locality can only be exploited with a fair, round-robin scheduler if the memory access instructions are issued at approximately similar times across the different warps. However, after the Adapt phase, there is a *skew* in the instruction issue pattern, as shown in Figure 10(a). Since GTO is used as the initial warp scheduler, if the round-robin scheduler is used immediately, instructions at different stages of the kernel will be issued across the different warps and the full benefit of a fair, round-robin scheduler will not be realized.

As a result, for workloads which need to be switched to a fair, round-robin scheduler, a Recover phase is necessary such that instructions issued among the different warps are balanced to fully realize the benefit of fair warp scheduling. During the recover phase, a *least instruction-issued* warp scheduler is used where the warps with the smallest number of instructions issued are prioritized. This warp scheduling during the Recover phase load-balances the instructions issued across the different warps. The Recover phase warp scheduling is done until the newest warp, which had the least number of instruction issued after the Adapt phase, finishes execution – i.e., matches the number of instruction issued from the oldest warp. After the Recover phase, the number of instructions issued across all of the warps will be approximately similar, as shown in Figure 10(b), and execution continues with a fair, round-robin scheduling until the end of the kernel.

The evaluation to determine the optimal scheduler is repeated for each new kernel. As we discuss later in Section 5, the different kernels within a given workload often have similar behavior but for some of the workloads (especially KMN), the different kernels can have different characteristics and the proposed iPAWS scheduler properly identifies the appropriate scheduler for each kernel. The benefits of the proposed iPAWS adaptive scheduler can be negligible if the ADAPT phase represents a significant portion of the execution of a kernel within a workload. This can occur if the amount of warps (or thread blocks) allocated to each core for a kernel are relatively small but for most of the GPGPU workloads, this is not necessarily the case. We show later in Section 4.1 that for the workloads that we evaluate, the ADAPT phase represents less than 10% of total execution time for most of the workloads.

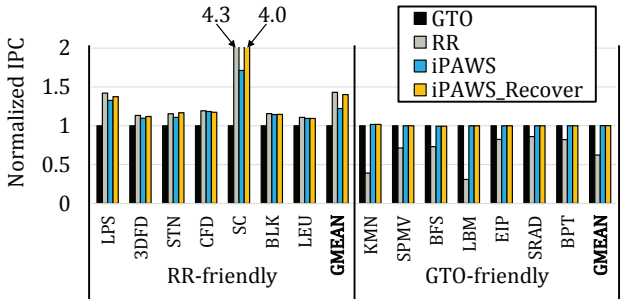

Figure 11: Performance of iPAWS adaptive scheduler and the impact of the recover phase.

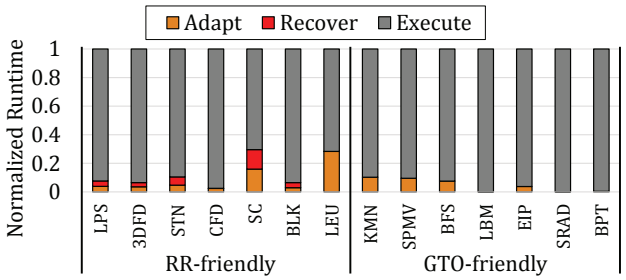

Figure 12: Overhead of the different phases in the adaptive scheduler.

3.5 Complexity

The main cost associated with the proposed iPAWS adaptive warp scheduler is the additional storage required to store the instruction-issue score. To record the instruction-issue score per warp, each core needs a set of counters that is equal to the maximum number of warps per core. The maximum number of warps per core in the latest NVIDIA GPU architecture such as Fermi and Kepler is 48 or 64. In this evaluation, we assume 48 warps per core and with 32-bit counters, the overhead is only 192 bytes of storage for each core and another set of counters is necessary for the instruction issue-stall. While 384 bytes of counter is not a significant overhead, the cost of the counters can be further reduced if necessary with minimal impact on performance by implementing a coarse-grained iPAWS where some number of warps share a single counter. We evaluated an iPAWS implementation where a single set of counter was used for each thread block and were able to reduce the amount of storage by a factor of 6 with no impact on overall performance.

4. EVALUATION

In this section, we present results and analysis for the proposed iPAWS. We then show how iPAWS can be extended to adapt between different warp schedulers as well as how it can be applied to GPGPU which support concurrent kernel execution.

4.1 Baseline iPAWS

The results of the proposed iPAWS adaptive scheduler is shown in Figure 11. We compare the results of a round-robin (RR) scheduler and the GTO scheduler with the proposed iPAWS, with and without the `Recover` phase. For all of the workloads, iPAWS is able to properly determine the more optimal warp scheduler and nearly match the performance of the more optimal warp scheduler. On the RR-friendly workloads, iPAWS without `Recover` phase achieves performance that is within 85% of the RR warp scheduler, on average, while exceeding GTO warp scheduler by 69%. However, for some of the workloads (e.g., `SC`), there is significant performance degradation from iPAWS without `Recover` phase compared to the RR scheduler. `SC` is a workload that has significant amount of inter-locality and with the imbalance from the initial GTO scheduler, the full benefit of RR warp scheduling is not realized. As a result, the `Recover` phase has significant impact on overall performance and increase `SC` performance by 2.3×. The performance difference between RR and GTO for `SC` is much greater than other RR-friendly workloads. The reason for this great difference is that all threads should access the same center point data to calculate the distance for its clustering algorithm. In other words, there exists inter-warp locality for this small amount of data across all threads in this workload. If we use RR scheduler, warps allocated to the same core can well exploit this locality in the L1 cache. On the other hand, with GTO scheduler, this data is frequently missed in the L1 cache of all cores and results in significant contention to a small portion of L2 cache and device memory. Thus, `SC` performance can be very lower with GTO than with RR scheduler. On average, iPAWS_`Recover` improves performance by 15% compared to iPAWS without `Recover` phase. For the rest of the evaluations, we assume iPAWS includes the `Recover` phase.

For the GTO-friendly workloads, iPAWS matches the performance of the GTO scheduler. Since iPAWS properly determines the more optimal scheduler and the initial warp scheduler was GTO, there is no performance degradation compared to GTO. However, iPAWS exceeds the performance of RR warp scheduler by 38% since RR scheduling cannot exploit the *intra*-warp locality or other benefit from a greedy scheduler. The `Recover` phase also has no impact on GTO-friendly workloads. The overhead from the `Adapt` and `Recover` phase is shown in Figure 12. Both the `LEU` and `SC` had the highest overhead as these two phases represented over 30% of the total execution time while for the other workloads, the overhead was less than 10%. Since the performance difference between RR and GTO was relatively small for `LEU`, this overhead had small impact on overall performance. For `SC`, a significant portion of execution time was required for the `Recover` phase and thus, iPAWS resulted in approximately 9% performance degradation, compared with the static, RR scheduler. Although not shown, the iPAWS has insignificant impact on performance for scheduler-insensitive workloads that were shown in Table 2. For these workloads, the fair and the greedy scheduler had minimal impact on performance as some of these workloads are compute-intensive while others have too simple work to be differentiated by warp scheduler.

The instruction-issue patterns for the workloads evaluated are shown in Figure 13 and the warps-of-interest (WOI) identified is also shown for each workloads. The area under the dotted line represents the threshold instruction score ($iscore_{thr}$) – thus, RR-friendly workloads have convex pattern ($iscore_{sum}$ is larger than $iscore_{thr}$) while the GTO-friendly

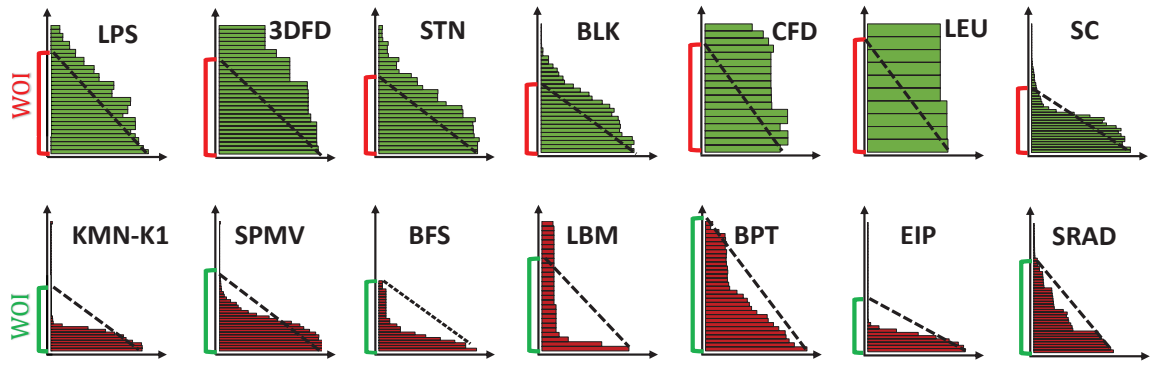

Figure 13: Instruction-issue patterns for the workloads evaluated. The top row shows RR-friendly workloads while the bottom row shows GTO-friendly workloads. Each plot is annotated with the warps-of-interest (WOI).

Table 3: The warps-of-interest (WOI) for evaluated workloads on iPAWS RR-GTO.

Name	MAX	WOI	Name	MAX	WOI
LPS	32	24	KMN-K1	48	24
3DFD	32	22	SPMV	48	32
STN	32	17	BFS	48	32
BLK	32	18	LBM	28	18
CFD	18	14	BPT	32	32
LEU	10	9	EIP	40	21
SC	48	24	SRAD	48	42

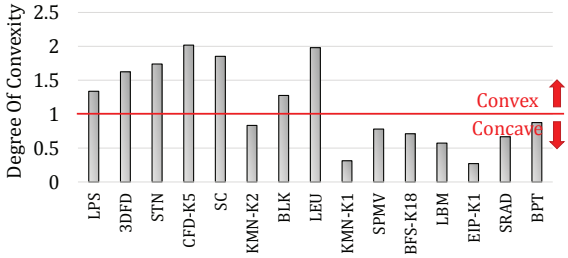

Figure 14: Degree of Convexity (DoC) when determining the more optimal warp scheduling.

workloads have concave pattern ($iscore_{sum}$ is smaller than $iscore_{thr}$). Table 3 also shows the WOI as well as the maximum number of warps allocated to each core. Most of the workloads have WOI that is smaller than the maximum number of warps allocated to the core. The ratio of the threshold instruction score ($iscore_{thr}$) and the sum of all the instruction score within the WOI ($iscore_{sum}$) are shown in Figure 14 – which we refer to as degree of convexity (DoC). For all of the convex workloads that we evaluate, $iscore_{sum}$ value is higher than the $iscore_{thr}$ by at least over 30%. For the concave workloads, $iscore_{sum}$ value is smaller than the $iscore_{thr}$ by at least over 15%. Although not shown, scheduler neutral workloads (MUM, RAY, etc) have a DoC ratio that is very close to 1. These workloads are not strictly convex or concave but since these workloads are compute intensive or cache-insensitive workloads, the warp scheduler does not have a significant impact on overall performance. For workloads with multiple kernels, only a representative kernels are shown in the figure, denoted with K*i* where *i* represents the *i*th kernel for the workload. Note that different kernels within a single workload can have different ratios and as we show in the following section, this can result in different optimal scheduling within a given workload – i.e., RR might be more optimal for one particular kernel while GTO might be more optimal for another kernel in the same workload.

4.2 Extending iPAWS to other Warp Schedulers

The adaptive scheduler proposed in this work was based on adapting between a fair, round-robin scheduler (RR) and a simply, greedy scheduler (GTO). However, this work is not necessarily limited to these two schedulers but other schedulers can be used with a similar adaptive decision. For example, since CCWS [20] is based on GTO, the GTO warp scheduler in iPAWS can be replaced with CCWS and the same instruction-issue pattern-based adaptive scheduler described earlier can be used. Figure 15 shows the result of iPAWS that adapts between RR and CCWS with the results normalized to the performance of GTO. For both types of workloads, iPAWS can properly identify the instruction-issue pattern and approach the performance of the more optimal warp scheduler. The performance of iPAWS with RR-CCWS matches the performance of CCWS for greedy-friendly workloads while for RR-friendly workloads, iPAWS exceeds the performance of CCWS by 40% on average. For GTO-friendly workloads with intra-warp locality, the instruction-issue pattern for CCWS shows a *more* concave pattern since CCWS often reduces the number of warps to issue their instructions while the other warps are throttled. In comparison, for RR-friendly workloads, the instruction-issue pattern is similar to the baseline GTO warp scheduler since CCWS behaves like GTO if intra-warp locality is not detected. Similar to CCWS, we also extend iPAWS to MASCAR [22] as well. Because of space constraints, the results for iPAWS that adapts between RR and MASCAR are not shown; however, the trend is similar to RR-CCWS adaptive scheduler and iPAWS is able to properly identify RR workloads for RR-friendly workloads while for other workloads, MASCAR was adaptively chosen.

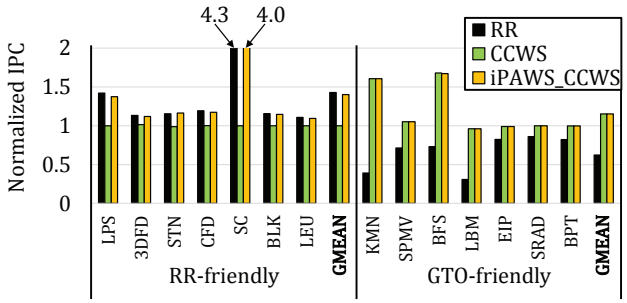

Figure 15: Evaluation of iPAWS that adapts between RR and CCWS.

4.3 Concurrent Kernel Execution

Modern GPU architectures support concurrent kernel execution (CKE) where multiple kernels are executed at the same time [17, 19]. CKE can be implemented in two different ways – partitioned concurrent kernel execution (pCKE) where cores are partitioned to different kernels [1, 3, 2] (i.e., a single core only executes one type of kernel) and shared concurrent kernel execution (sCKE) where multiple kernels run on the same core at the same time [17, 19]. In this section, we show how iPAWS can be leveraged for both types of concurrent kernel execution.

In our baseline iPAWS, since all of the cores are executing the same kernel, statistics from only one core was necessary to determine the more optimal warp scheduler. To extend iPAWS to pCKE, the only extension necessary is that each core needs to independently determine the more optimal warp schedulers since different kernels can have different instruction-issue patterns. Aside from this change, the same warp scheduler can be used for pCKE to optimize each kernel separately (in parallel) to maximize overall performance. Since most of the applications used in our evaluation do not exploit CKE, we merged different workloads (Table 4) and evaluated iPAWS. The results of iPAWS on pCKE is shown in Figure 16(a). For four out of the five workload mixes, each kernel has a different more optimal warp scheduler and thus, a single warp scheduler across all of the cores (either RR or GTO) does not provide high performance. However, by leveraging iPAWS, each core is able to properly identify the instruction-issue pattern for the different kernels and iPAWS is nearly able to match the performance of GTO-RR – a statically determined scheduler where the GTO-friendly workload is executed by the GTO warp scheduler and the RR-friendly workload uses the RR warp scheduler.

However, iPAWS cannot be directly used for sCKE where different kernels share the same core since iPAWS relies on properly determining the warps-of-interest (WOI). Since the core resources are being shared, the warp instruction issue patterns can be impacted by the concurrent kernel on the same core. To minimize the impact of concurrent kernels on the adaptive decision, we group two cores (e.g., core0 and core1) together such that each core can be used separately to determine the more optimal warp scheduler for two kernels (e.g., K0 and K1). In core0, we assume warps from K0 are "older" and are prioritized over warps from K1. Similarly, the opposite is done within core1 – warps from K1 are prioritized

Table 4: Different combinations of workloads for CKE evaluation and their more optimal scheduler combination.

Workload combination	Optimal scheduler combination
KMN-BLK	GTO-RR
KMN-SC	GTO-RR
LBM-BLK	GTO-RR
LBM-SC	GTO-RR
KMN-LBM	GTO-GTO

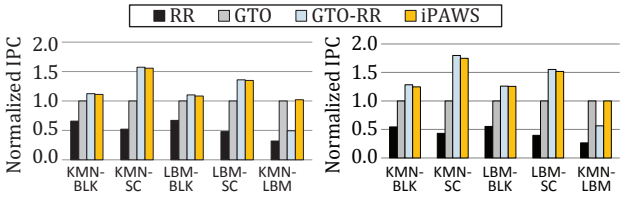

Figure 16: Results of iPAWS with (a) partitioned and (b) shared concurrent kernel execution.

over K0. In determining the WOI, only the warps from K0 are considered for WOI in core0 and warps from K1 are considered for WOI in core1. The results for iPAWS with sCKE is shown in Figure 16(b) and similar to pCKE, iPAWS was able to identify the more optimal warp scheduler for each kernel. In our evaluation of sCKE, we assumed that the resources (e.g., register file, shared memory) in each core are shared equally between the two kernels.

5. DISCUSSION

In this section, we first explore the impact of different kernels *within* a workload and how adaptive scheduling impacts overall performance. In particular, we show how a static approach of determining the optimal scheduler (e.g., profiling-based approach) is not necessarily accurate since the workload behavior can change based on the workload input. We also provide a discussion on an alternative adaptive warp scheduling and the impact of using the first warp as a metric within iPAWS.

5.1 Inter-Kernel Variation

While most of the workloads that we evaluated often tend to repeat kernels that have similar characteristics, KMN was one workload where the two kernels (k0 and k1) had different locality characteristics (Figure 17) – i.e., k0 has intra-warp locality while k1 is dominated by inter-warp locality. [2] As a result, our adaptive warp scheduler is able to dynamically determine the optimal scheduler for each kernel – GTO scheduler for k0 and RR scheduler for k1. Figure 18 shows the different types of locality for the two kernels (k0, k1) of KMN. When GTO is used instead of RR, inter-warp locality in both L1 and L2 cache is decreased for kernel k1 while intra-warp locality in L1 cache is dramatically increased for kernel k0. As we discuss in the following section, for some

[2]In KMN, k0 inverts an array of multi-dimensional points from point-major order to dimension-major order, and then k1 finds the nearest cluster from the inverted array [5].

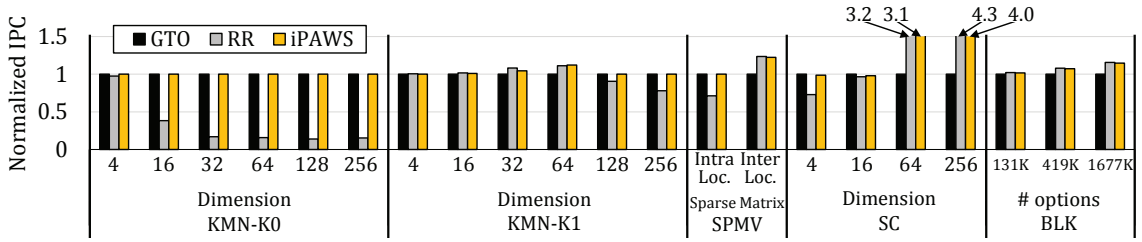

Figure 17: Impact of different warp scheduler as workload input and parameter is changed.

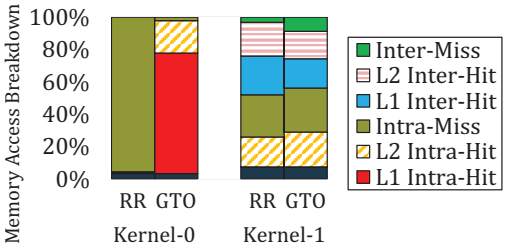

Figure 18: Memory access breakdown for kernel-0 and kernel-1 of KMN.

input dimension of KMN (e.g., 32 or 64), iPAWS can *exceed* the performance of the statically-determined GTO scheduler by approximately 10% because of the ability of iPAWS to adapt appropriately for each kernel.

5.2 Workload Input Variation

In this section, we discuss the impact of workload inputs and the impact of warp scheduling. A single set of workload inputs is commonly used to evaluate alternative warp schedulers. However, as shown in Figure 17, different workload inputs can impact the performance of the warp schedulers. For some workloads, the workload input changes the performance gap between RR and GTO warp scheduler while for other workloads, the optimal scheduler can change.

For example, SC exploits inter-warp locality when the coordinates of cluster center is accessed by different warps while intra-warp locality is exploited when a warp accesses its own data points (e.g., cost, weight) repeatedly. If the dimension of each point is sufficiently large (D $\geq$ 16), the coordinate data of the cluster center point is more frequently accessed and thus increasing inter-warp locality while the amount of intra-warp locality remains similar. On the other hand, if each data point has small dimension, the more optimal scheduler is GTO since the amount of inter-warp locality is highly reduced while intra-warp locality is more dominant. For SPMV, the optimal warp scheduler can also change based on the input sparse matrix as shown in Figure 17. An input matrix where non-zero elements tend to be adjacent in the same column rather than the same row (Inter Loc.) is more suitable for inter-warp locality for the SPMV [24] implementation. On the other hand, if non-zero elements are adjacent in the same row of a matrix (Intra Loc.), intra-warp can be better exploited.

For some workloads, it might be practical to implement multiple warp schedulers and have the compiler provide hints on the more optimal warp scheduler. While this approach might work for some workloads, the different workloads shown in Figure 17 likely will not be properly identified by the compiler. In comparison, iPAWS is able to properly adapt to the appropriate warp scheduler.

KMN is another workload where behavior can change significantly based on the workload input. For the first kernel (KMN-K0), the amount of data processed by each warp increases as the dimension size increases and locality cannot be fully captured by the L1 data cache. A fair, RR scheduler results in continuous capacity misses while a greedy scheduler better exploits the intra-warp locality. Thus, GTO scheduler achieves significantly higher performance compared to the RR scheduler. However, for the second kernel (KMN-K1), the RR provides better performance at 32-64 dimensions and then, decreases in performance. Based on the analysis, RR warp scheduling better exploits the texture cache but as the dimension increases, the texture cache hit rate decreases and thus, greedy approach provides slightly better performance.

5.3 Locality-based Adaptive Scheduling

Since the cache locality is an important factor of adaptive warp scheduling, we explore the trade-off of implementing a locality-based adaptive scheduling compared with iPAWS. We implement a locality-based adaptive scheduling (Adapt$_{local}$) that compares the amount of inter-warp and intra-warp locality to adapt between two warp schedulers. We extend the intra-warp locality detection mechanism used for CCWS [20] to measure inter-warp locality both in the L1 and the L2 cache. We then compare the amount of inter-warp locality and intra-warp locality at the end of each sampling period where the sampling period was empirically determined to provide the highest performance. [3] Figure 19 shows the performance of locality-based adaptive scheduling (Adapt$_{local}$) and compares it with iPAWS. Similar to iPAWS, we assume the baseline warp scheduler is GTO for Adapt$_{local}$. For workloads where locality had significant impact (LPS, 3DFD, CFD, SC, KMN, BFS, LBM, EIP), Adapt$_{local}$ is able to adapt to the appropriate warp scheduler and match the performance of iPAWS. For some workload such as SC, the performance of Adapt$_{local}$ exceeds that of iPAWS since Adapt$_{local}$ does not have the overhead of the `Adapt` phase.

However, for other workloads (BLK, LEU, BPT, and SRAD), Adapt$_{local}$ is not able to properly identify the more optimal warsince these workload behavior do not depend on cache locality. In addition, for some workloads (SPMV and STN), simply comparing both types of locality quantitatively fails to find the optimal scheduler. Even though the impact of warp scheduler is influenced by one type of locality (e.g.,

[3] Short sampling period did not provide sufficient amount of time to properly adapt while longer sampling period resulted in high overhead, similar to the `Adapt` phase of our proposed iPAWS.

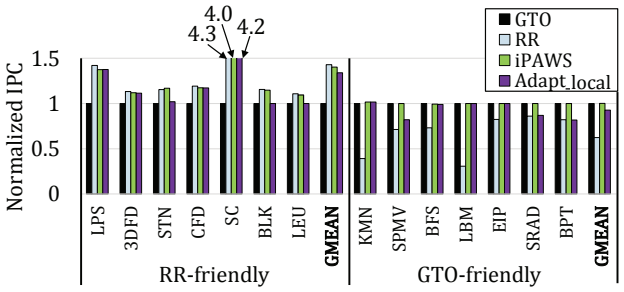

Figure 19: Comparison with the locality-based adaptive scheduling.

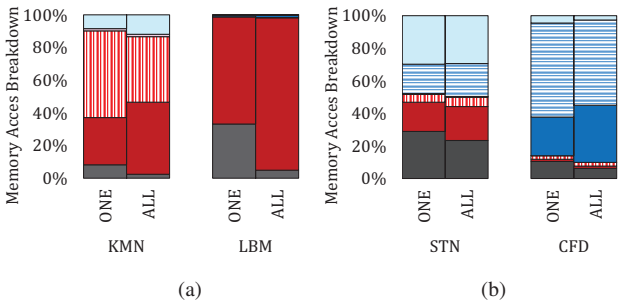

Figure 20: Cache access breakdown for (a) GTO-friendly workloads and (b) RR-friendly workloads until the first warp terminates (ONE) and the kernel terminates (ALL).

intra-warp locality for SPMV and inter-warp locality for STN), both types of locality exist for these workloads. Thus, the adaptive warp scheduler based strictly on locality information cannot necessarily determine the more optimal warp scheduler if the locality quantity information between the two types of locality is similar. As a result, one challenge with Adapt$_{local}$ is to determine the empirical thresholds for each workload such that proper weights are added to each component of locality. In addition, implementation overhead of Adapt$_{local}$ is higher – i.e., a victim list to store warp and core IDs for each cache line of both L1 and L2 is needed.

5.4 Impact of Using First Warp Statistics

Our proposed iPAWS adaptive scheduler determined the instruction-issue pattern after the *first* warp finished execution. The earlier locality analysis in Figure 4 was done after the first warp finished execution, but in Figure 20, we also show locality information when it is collected after *all* warps within the kernel finish execution. Only a few representative workloads are shown and locality information can change – i.e., the ratio of cold misses decrease since the workload is executed longer. However, the fundamental behavior (i.e., inter- vs intra-warp locality) of the workload does not change during the execution of the entire kernel.

In our implementation of iPAWS, we also used the adaptive decision results from the first core that finishes execution. We also compared iPAWS with an implementation where each core determines the warp scheduler independently. For this implementation (iPAWS$_{ind_cores}$), there was no impact on the performance for GTO-friendly workloads but for RR-friendly workloads, there was actually some small performance degradation. For SC, iPAWS$_{ind_cores}$ resulted in approximately 10% performance degradation, compared with the original iPAWS. Since the execution time of each core can vary, the Adapt phase of each core will also vary and some of the cores that are stalled more than others will end up spending more time in the Adapt. This delays the switch to the RR warp scheduler for some of the cores and degrades overall performance. Thus, for a single kernel, using the workload behavior during first warp as well as the information from a single core is sufficient to properly identify the workload characteristics.

6. RELATED WORK

There have been various warp schedulers proposed which are based on a *greedy* approach of prioritizing certain warps over others warps. In addition, GTO and CCWS [20], Divergence-Aware Warp Scheduling (DAWS) [21] also track L1 cache accesses to pro-actively throttle the number of warps scheduled. Lazy CTA Scheduler [12] also leveraged GTO warp scheduler to find the optimal number of CTAs and avoid performance degradation. MASCAR [22] proposed a greedy scheduling technique to detect memory saturation and limit warps which send memory requests simultaneously.

Hierarchical warp scheduling or two-level warp scheduling (TLV) was first proposed in [15] to improve latency hiding capability. TLV groups warps into several fetch groups and prioritizes one fetch groups until all warps in the fetch groups are stalled. TLV-based schedulers include CTA-aware scheduling (OWL) [8] and prefetch-aware warp scheduling (PA) [9]. OWL prioritizes a group of CTAs to improve performance while PA groups non-consecutive warps to enable effective prefetching. PATS [26] proposed warp scheduling to effectively power-gate SIMT lanes based on branch divergence and focuses on energy-efficiency with minimal performance degradation. Song et al. [23] also propose a throttle thread block scheduling to improve the energy-efficiency of GPGPUs. CAWS [13] proposes a family of criticality-aware warp schedulers to compensate latency divergences among warps. This work proposes an adaptive approach that can be exploited with other prior work on schedulers to adapt as necessary between alternative schedulers in GPGPU.

To efficiently utilize L1 cache, L1 cache bypassing at compile-time has been proposed [7]. On the other hand, we do not focus on explicit cache management but focus on adaptive scheduling at run-time. Memory Request Prioritization Buffer (MRPB) [25] was proposed to reduce L1 cache contention by reordering requests and bypassing L1 cache. Our adaptive scheduling can complement MRPB by selecting optimal warp scheduler for a given workload. Kaleem *et al.* [10] proposed adaptive scheduling techniques for integrated CPU-GPU processors. They focused on partitioning the work of data-parallel kernels between CPU and GPU.

Contemporary GPUs [17, 19] support the concurrent kernel execution (CKE) which allows independent kernels to be executed at the same time. Adriaens *et al.* [1] proposed spatial multi-tasking where multiple applications partition cores among the applications running concurrently. Liang *et al.* [14] proposed a SW/HW solution which allows multiple applications to share the GPU temporally and spatially. Aguilera *et al.* [3] proposed a runtime technique to partition GPU resources dynamically to maximize the benefit of spatial multitasking for QoS. For spatial multitasking, Fair Share [2]

observed the unfair access to GPU resources among applications. They also proposed to characterize fairness and balance performance of each application and overall performance for GPU spatial multitasking.

7. CONCLUSION

While there has been a significant amount of research done on thread (or warp) scheduling in manycore accelerators such as GPGPUs, prior works have mostly assumed *static* scheduling as a single warp scheduler is used for all of the workloads. However, we show that a single warp scheduler is not necessarily optimal across different GPGPU workloads. Thus, we propose an *adaptive* warp scheduler that dynamically adapts between two different schedulers based on the runtime characteristics of the workloads. In particular, we propose iPAWS (Instruction-Issue Pattern-based Adaptive Warp Scheduler) that adaptively determines whether a greedy warp scheduler (GTO) or a fair, round-robin scheduler is more optimal based on the instruction-issue pattern across the different warps within the core. We show how workloads that favor greedy warp scheduler tend to have a *concave* instruction-issue pattern shape while workloads that favor a fair, round-robin warp scheduler have a *convex* instruction-issue pattern. By properly identifying the instruction-issue pattern, we show that iPAWS can not only differentiate workloads with intra- and inter-warp locality but also properly adapt for workloads that do not have such locality. While there have been various warp schedulers proposed to improve performance for different workloads, this work complements those work by showing that an *adaptive* approach to warp scheduling can provide better performance across different types of workloads.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. This research was supported in part by the Mid-career Researcher Program (NRF-2013R1A2A2A01069132), in part by the MSIP under the ITRC support program (IITP-20 15-H8501-15-1005) supervised by the IITP, and in part by Next-Generation Information Computing Development Program through NRF funded by the Ministry of Science, ICT & Future Planning (2015M3C4A7065647).

8. REFERENCES

[1] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The Case for GPGPU Spatial Multitasking," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, New Orleans, LA, 2012, pp. 1–12.

[2] P. Aguilera, K. Morrow, and N. S. Kim, "Fair share: Allocation of GPU resources for both performance and fairness," in *Intl. Conf. on Computer Design (ICCD)*, Seoul, Korea, 2014, pp. 440–447.

[3] P. Aguilera, K. Morrow, and N. S. Kim, "QoS-Aware Dynamic Resource Allocation for Spatial-Multitasking GPUs," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Singapore, 2014, pp. 726–731.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads using a Detailed GPU Simulator," in *Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, Boston, Massachusetts, 2009, pp. 163–174.

[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 *Intl. Symp. on Workload Characterization (IISWC)*, Austin, TX, 2009, pp. 44–54.

[6] 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 *Intl. Symp. on Computer architecture (ISCA)*, San Jose, California, 2011, pp. 235–246.

[7] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-fetched Caches in GPUs," in *Int. Conf. on Supercomputing (SC)*, Venice, Italy, 2012, pp. 15–24.

[8] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, 2013, pp. 395–406.

[9] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Intl. Symp. on Computer Architecture (ISCA)*, Tel-Aviv, Israel, 2013, pp. 332–343.

[10] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive Heterogeneous Scheduling for Integrated GPUs," in *Intl. Conf. on Parallel Architectures and Compilation*, Edmonton, AB, Canada, 2014, pp. 151–162.

[11] Khronos OpenCL Group, "The OpenCL Specification," 2011.

[12] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization through Alternative Thread Block Scheduling," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Orlando, Florida, 2014, pp. 260–271.

[13] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *Int. Conf. on Parallel Architectures and Compilation (PACT)*, Edmonton, AB, Canada, 2014, pp. 175–186.

[14] Y. Liang, H. Huynh, K. Rupnow, R. Goh, and D. Chen, "Efficient GPU Spatial-Temporal Multitasking," *Parallel and Distributed Systems, IEEE Transactions on (TPDS)*, vol. 26, no. 3, pp. 748–760, 2015.

[15] 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 *Intl. Symp. on Microarchitecture (MICRO)*, Porto Alegre, Brazil, 2011, pp. 308–317.

[16] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011.

[17] NVIDIA, "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," 2011.

[18] NVIDIA, "CUDA C Programming Guide," 2012.

[19] NVIDIA, "Kepler: The Fastest, Most Efficient HPC Architecture Ever Built," 2012.

[20] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Intl. Symp. on Microarchitecture (MICRO)*, Vancouver, B.C., Canada, 2012, pp. 72–83.

[21] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware Warp Scheduling," in *Intl. Symp. on Microarchitecture (MICRO)*, Davis, California, 2013, pp. 99–110.

[22] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up GPU Warps by Reducing Memory Pitstops," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, San Francisco Bay Area, CA, 2015, pp. 174–185.

[23] S. Song, M. Lee, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Energy-efficient scheduling for memory-intensive GPGPU workloads," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, Dresden, Germany, 2014, pp. 1–6.

[24] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," no. IMPACT-12-01, 2012.

[25] M. M. Wenhao Jia, Kelly A. Shaw, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Orlando, FL, 2014, pp. 272–283.

[26] Q. Xu and M. Annavaram, "PATS: Pattern Aware Scheduling and Power Gating for GPGPUs," in *Intl. Conf. on Parallel Architectures and Compilation (PACT)*, Edmonton, AB, Canada, 2014, pp. 225–236.