CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads

Shin-Ying Lee Akhil Arunkumar Carole-Jean Wu
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
{lee.shin-ying,akhil.arunkumar,carole-jean.wu}@asu.edu

Abstract

The ubiquity of graphics processing unit (GPU) architectures has made them efficient alternatives to chip-multiprocessors for parallel workloads. GPUs achieve superior performance by making use of massive multi-threading and fast context-switching to hide pipeline stalls and memory access latency. However, recent characterization results have shown that general purpose GPU (GPGPU) applications commonly encounter long stall latencies that cannot be easily hidden with the large number of concurrent threads/warps. This results in varying execution time disparity between different parallel warps, hurting the overall performance of GPUs – the warp criticality problem.

To tackle the warp criticality problem, we propose a coordinated solution, criticality-aware warp acceleration (CAWA), that efficiently manages compute and memory resources to accelerate the critical warp execution. Specifically, we design (1) an instruction-based and stall-based criticality predictor to identify the critical warp in a thread-block, (2) a criticality-aware warp scheduler that preferentially allocates more time resources to the critical warp, and (3) a criticality-aware cache reuse predictor that assists critical warp acceleration by retaining latency-critical and useful cache blocks in the L1 data cache. CAWA targets to remove the significant execution time disparity in order to improve resource utilization for GPGPU workloads. Our evaluation results show that, under the proposed coordinated scheduler and cache prioritization management scheme, the performance of the GPGPU workloads can be improved by 23% while other state-of-the-art schedulers, GTO and 2-level schedulers, improve performance by 16% and -2% respectively.

1. Introduction

Graphics Processing Units (GPUs) have recently become an efficient alternative to traditional chip-multiprocessors (CMPs) for executing general purpose parallel workloads (GPGPU). Modern GPU architecture designs are based on the *Single Instruction Multiple Thread* (SIMT) computing paradigm where multiple threads are grouped together to form a *warp* or *wavefront*. Threads within a warp are mapped to a vector functional unit or *Single Instruction Multiple Data* (SIMD) unit such that all threads execute the same instruction, but with different data, simultaneously [22]. Furthermore, the concept of fine-grained multi-threading is implemented in modern GPUs by enabling a number of warps to be executed in a GPU core in the time-multiplexing manner. For example, NVIDIA's Maxwell GPUs support up to 64 concurrent warps (equivalent to 2048 threads) within a GPU streaming multiprocessor (SM) [27].

GPUs are able to hide the stall latency from a warp with useful instruction execution from another warp through fast context-switching—whenever the execution of a warp is stalled, it is swapped out and another warp will be swapped in for immediate execution to increase resource utilization. However, it has been recently shown that the latency hiding ability of GPUs has significant room for improvement [21]. The limited latency hiding ability of GPU schedulers is commonly caused by imbalanced workloads in parallel warps [20], diverging branch behavior [9, 10, 23, 32], irregular memory access patterns [4, 28], and shared cache contention [24, 34]. This could result in a significant execution time disparity between different parallel warps in many GPGPU applications—the *warp criticality* problem. All warps within a thread-block have to finish execution before the particular thread-block can commit and another thread-block can be dispatched to the SM. A recent prior work has performed a case study for an implementation of the breadth-first-search (`bfs`) algorithm, demonstrating the significant warp execution time gap between the fastest and the slowest (critical) warps to be over 50% [20]. If oracle knowledge of warp criticality were available, the Criticality-Aware Warp Scheduler (CAWS) could prioritize the critical warps over other faster running warps, resulting in 21% performance improvement. However, the CAWS requires oracle warp criticality information; therefore, their proposed solution is not practical and cannot adapt to runtime behavior.

To address the warp criticality problem, this paper proposes a runtime Coordinated criticality-Aware Warp Acceleration (**CAWA**) solution to reduce the execution time disparity between parallel warps. CAWA manages shared compute and memory resources on a GPU in order to accelerate the execution of slower-running warps. The proposed CAWA design

ISCA'15, June 13-17, 2015, Portland, OR, USA

http://dx.doi.org/10.1145/2749469.2750418

consists of three main components:

- **Dynamic Warp Criticality Prediction with $CAWA_{CPL}$:** To identify critical warps on the fly, we design the instruction-based and stall-based *criticality prediction logic* (CPL). $CAWA_{CPL}$'s warp criticality prediction outcome is used to guide shared resource prioritization between the faster and slower-running warps at runtime by the criticality-aware warp scheduler and criticality-aware cache management (Section 3.1).
- **Greedy Criticality-Aware Warp Scheduler with $CAWA_{gCAWS}$:** Building on top of two existing warp scheduling policies [20, 34], we propose a *greedy Criticality-Aware Warp Scheduler* (gCAWS) to preferentially allocate more time resources to slower-running warps for critical warp acceleration at the scheduling level (Section 3.2).
- **Criticality-Aware Cache Prioritization with $CAWA_{CACP}$:** To further accelerate the execution of critical warps, we propose a Criticality-Aware Cache Prioritization (CACP) scheme, which consists of a *Critical Cache Block Predictor* that predicts whether an incoming cache block should be inserted into the critical cache partition reserved for critical data in the L1 data cache and a modified *Signature-based Cache Hit Predictor* [38] that predicts the reuse pattern of an incoming cache block. $CAWA_{CACP}$ proactively predicts and retains data that is performance-critical with higher priorities—data useful to slower-running, critical warps and data that exhibits locality (Section 3.3).

The main concept of CAWA design is to manage both compute resources at the warp scheduler's level and memory resources at the L1 data caches to accelerate the execution of critical warps. By doing so, the large execution time gap between the fastest-running and the slowest-running warps in a thread block is significantly reduced leading to a much reduced synchronization overhead in the massively-parallel GPU architecture. Our evaluation results show that the performance of target GPGPU applications can be improved by an average of 23% and by as much as 3.13 times for `kmeans`.

In summary, this paper makes the following contributions:

- We present an in-depth characterization to quantify the degree of warp criticality caused by workload imbalance, diverging branch behavior, memory subsystem behavior, and the warp scheduler for GPGPU workloads.
- We propose a coordinated warp scheduling and cache prioritization scheme to accelerate the execution of critical warps in GPGPU workloads. This is the first work that offers a practical solution for the warp criticality problem with an accurate warp criticality prediction technique as well as a criticality-aware warp scheduler and a cache prioritization scheme.

The rest of the paper is organized as follows. Section 2 gives an overview of the warp criticality problem in GPGPUs and presents in-depth performance characterization results for understanding the sources of warp criticality. We present the proposed coordinated criticality-aware warp acceleration design in Section 3. Next, we show the experimental methodology in Section 4, followed by the evaluation results and analysis in Section 5. Section 6 presents related work in this area and Section 7 concludes the paper.

2. Background and Motivation

2.1. Background on GPU Architecture

A modern unified GPU consists of multiple streaming-multiprocessors (SMs) or computation units (CUs). An SM is similar to a Single Instruction Set Multiple Data (SIMD) processor, having one or multiple vector functional units, data caches, instruction caches, and instruction fetch/decode units. An SM also has a warp pool to store the context of all running threads. It implements a hardware warp scheduler that dispatches and allocates resources for thread execution, and has a large register file which is shared across all threads and accommodates the private data of all threads.

A GPGPU application comprises massive number of threads that process the same piece of code named a *kernel*. At run time, multiple threads form a *thread-block* or a *cooperative thread array* (CTA). A thread-block is the basic unit of GPGPU application execution. All threads within a thread-block have the same life-cycle and are dispatched onto as well as swapped out from an SM at the same time. Depending on the width of the vector functional units/SIMD lanes, threads within a thread-block are also grouped into small batches (typically 32 or 64 threads together) called a warp or a wavefront. At every cycle, the hardware warp scheduler selects a ready warp—a warp that does not stall due to factors such as cache misses or pipeline hazards—from the warp pool for execution.

With this massive multi-threading paradigm and the large register file design, whenever a thread/warp stalls, GPUs are able to quickly perform context-switching to hide the execution latency with minimal penalty. Hence, modern GPUs are able to achieve a higher average pipeline utilization and throughput compared to traditional CMP designs.

2.2. The Warp Criticality Problem

Warps within a thread-block are tightly coordinated. They are bound to the same synchronization barrier (either explicit or implicit) and have the same life-cycle. That is, warps from the same thread-block must start execution at the same time, be blocked at a synchronization barrier until all warps have arrived, and wait at the kernel exit point until all warps have finished execution. Nevertheless, not all warps finish their tasks at the same time. There is execution time disparity observed among warps [20]. Consequently, fast warps are idle at a synchronization barrier or at the end of kernel execution until the slowest warp arrives. This fact results in two problems which degrade a GPU's overall throughput.

- A thread-block's execution time is dominated by the slowest warp or the so-called *critical warp*. Although fast warps have finished all the tasks, they are suspended at an explicit (e.g., an synchronization instruction) or implicit (e.g., kernel exit point) synchronization barrier without

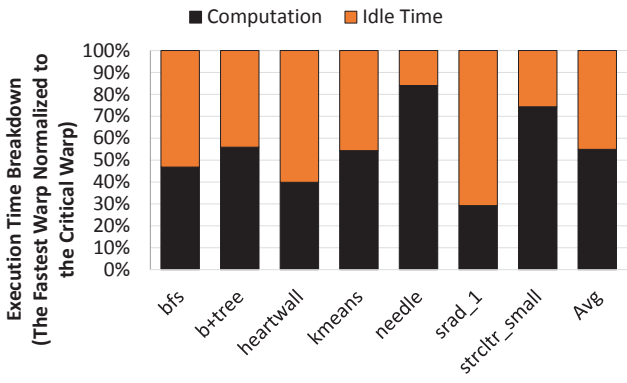

Figure 1: Warp execution time disparity across different GPGPU applications. The figure shows the highest warp execution time disparity across thread blocks.

doing any useful computation. As a result, it wastes lots of hardware resources such as the registers.

- When fast warps have finished execution and are in the idle state, the number of active warps decreases. In such a scenario, the GPU may not have sufficient warps to hide the execution latency. When a warp stalls, its execution latency will be exposed and thereby the GPU cannot achieve the maximum throughput.

We call this sub-optimal performance problem as the *warp criticality* problem. To highlight the importance of the warp criticality problem in GPGPU applications, we evaluate the execution time difference between the fastest and the slowest running warps in a thread-block. Figure 1 shows that the average execution time difference across a number of GPGPU applications is 45% and the difference could be as high as 70% (`srad_1`). Such significant execution time disparity between parallel warps is caused by four factors—(1) workload imbalance, (2) diverging branch behavior, (3) contention in the memory subsystem, and (4) warp scheduling order. We use `bfs` as an example application to illustrate the degree of warp criticality for each factor.

2.2.1. Workload Imbalance In a GPGPU kernel function, tasks are not always uniformly distributed to each thread/warp, and thereby some threads/warps have heavier workloads than others. Intuitively, the threads/warps with heavier workloads require longer time to process their tasks. Consequently, warps with heavier workloads often become the slowest-running/critical warps.

In `bfs`, workload imbalance comes from building and traversing an unbalanced tree data structure. Each node has to traverse all of its neighboring nodes to build a tree (line 2 in Algorithm 1). Depending on data inputs, the number of child nodes of a particular node mapped to a thread/warp could vary, resulting in different per-warp workloads. Figure 2(a) shows the per-warp execution time for all warps in a particular thread-block (Thread-Block 2 on SM 3) sorted based on warp execution time. The execution time disparity between the fastest and the slowest running warps is approximately 20% of the fastest warp's execution time, leading to a significant waiting time for the fastest running warp.

Algorithm 1 bfs searching algorithm

```
1:  function BFS(w)                          ▷ w: node (thread/warp)
2:      while notYetVisitedAllNeighbors do
3:          n ← nextNode
4:          if n.hasNotBeenVisited then      ▷ a child node
5:              n.Cost ← w.Cost
6:              n.hasNotBeenVisited ← False
7:              w.nChild ← w.nChild + 1
8:          else                             ▷ a non-Child node
9:              w.nNonChild ← w.nNonChild + 1
10:         end if
11:     end while                            ▷ kernel exit point/implicit barrier
12: end function
```

2.2.2. Diverging Branch Behavior In addition to the unbalanced workload scenario for parallel warps in a thread block, diverging branch behavior could also cause varying execution time for warps. At runtime, warps can undergo different control paths leading to different number of dynamic instructions across different warps. This problem could be worsened if threads in a warp also take diverging control paths, i.e., the branch divergence problem, leading to a larger instruction execution gap between warps. Prior studies [9, 10, 23, 32] showed that the branch divergence problem can significantly degrade the performance of GPGPU applications.

To remove the workload imbalance effect and focus on the impact of the diverging branch behavior, we modify the data input provided to `bfs` to represent a balanced tree. Figure 2(b) shows the warp execution time for the same thread-block with a balanced workload. While the computation workload is equally distributed across warps, we observe varying warp execution time. The execution time difference between the fastest and the slowest warps is significant 40%. This is because while a node traverses through its neighbors in the input graph, only the data stored in child nodes (nodes that have not yet been visited) need to be processed. Visiting child and non-child nodes (nodes that have been visited before) fall onto different if-else blocks (lines 4–10 in Algorithm 1). This introduces a varying number of per-warp dynamic instruction counts, since some warps will execute the taken path while others execute the not-taken path.

In the worst-case scenario, when thread-level branch divergence occurs [23, 32], instructions in both the taken and not-taken paths need to be executed in some warps resulting in a higher number of dynamic instruction executed, while other warps only have to execute one of the two paths. The dynamic instruction count disparity between warps could be as high as 20% (number of instructions) in `bfs` as illustrated by the red curve in Figure 2(b).

2.2.3. Contention in the Memory Subsystem Hardware resource contention, particularly in the memory subsystem, can exacerbate the warp criticality problem. Jog et al. observed that the memory subsystem has a significant impact on GPGPU applications [15, 16]. Poor data alignment and

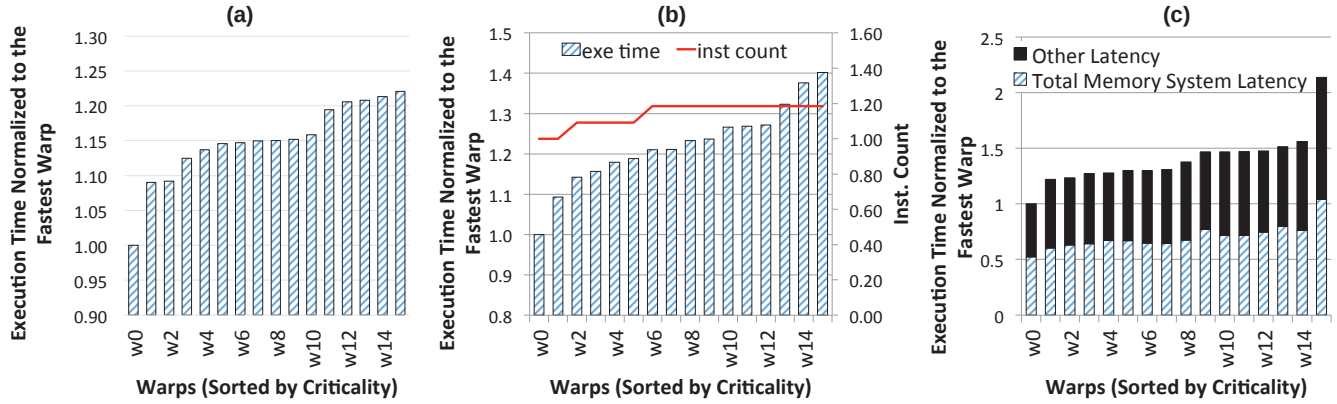

Figure 2: Warp execution time disparity caused by (a) workload imbalance, (b) diverging branch behavior, and (c) additional memory subsystem delay for `bfs`.

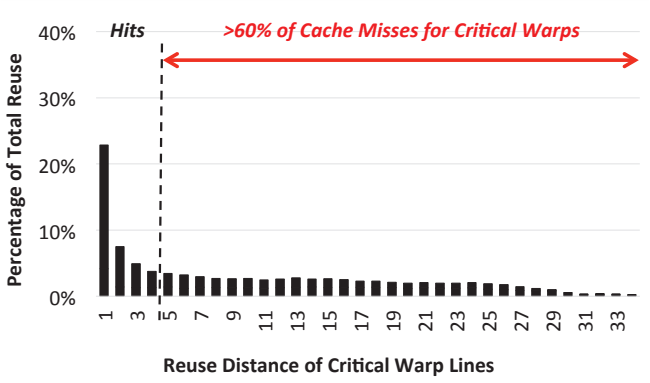

Figure 3: Reuse distance analysis for the critical warps in *bfs*.

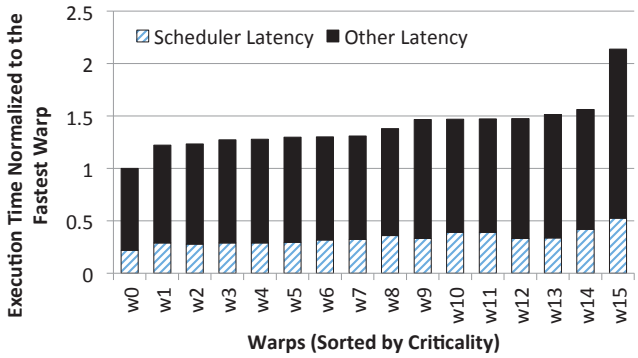

Figure 4: Warp scheduler could introduce additional stall cycles to warps.

warp scheduling design can introduce extra stall cycles which significantly reduce the performance of GPUs. Jia et al. also pointed out that interference in the L1 data cache as well as in the interconnection between the L1 data caches and the L2 cache are the major factors that limit GPU performance [13, 14]. This is because the data cache capacity and the memory bandwidth are scarce resources servicing the memory demand of the massively parallel GPUs.

The cache interference, in particular in the L1 data caches, could worsen the warp criticality problem. Figure 2(c) shows the portion of a warp's execution time caused by delays in the memory subsystem. We can observe that the slower-running warps often also experience higher memory access latencies. Figure 3 shows the reuse distance analysis of critical warp cache lines[1]. More than 60% of the cache blocks that could be reused by the slower-running, critical warps are evicted before the re-references by the critical warps. This is caused by the interference between critical and non-critical cache blocks in the L1 data cache.

2.2.4. Latency Introduced by the Warp Scheduler The execution of warps can experience additional delay in the warp scheduler. Because of the particular warp execution order determined by the scheduler, when a warp becomes ready for execution, it can experience up to N cycles of scheduling delay, where N represents the number of warps. State-of-the-art warp scheduling policies are criticality-oblivious and introduce additional delay that could further degrade the performance of critical warps. As Figure 4 shows, scheduling policies, such as the baseline round robin scheduler, can contribute as much as 52.4% additional wait time for the critical warp.

[1]Data is based on an L1 data cache of 16KB, 4-way set-associative, 128B cache block.

3. CAWA: Coordinated Criticality-Aware Warp Acceleration

We propose a coordinated warp scheduling and cache prioritization solution, called **C**oordinated criticality-**A**ware **W**arp **A**cceleration (**CAWA**). CAWA consists of three major components: (1) a dynamic warp criticality predictor, (2) a greedy criticality-aware warp scheduler, and (3) a criticality-aware cache prioritization technique. Figure 5 illustrates the different components in CAWA in the context of a modern GPU pipeline (NVIDIA Fermi architecture).

CAWA identifies slower-running warps that have a high likelihood to become critical warps at runtime ($CAWA_{CPL}$). CAWA allocates more compute resources to the predicted-to-be-critical warps with higher scheduling priorities ($CAWA_{gCAWS}$). This alleviates the dynamic workload imbalance-caused warp criticality as well as the additional

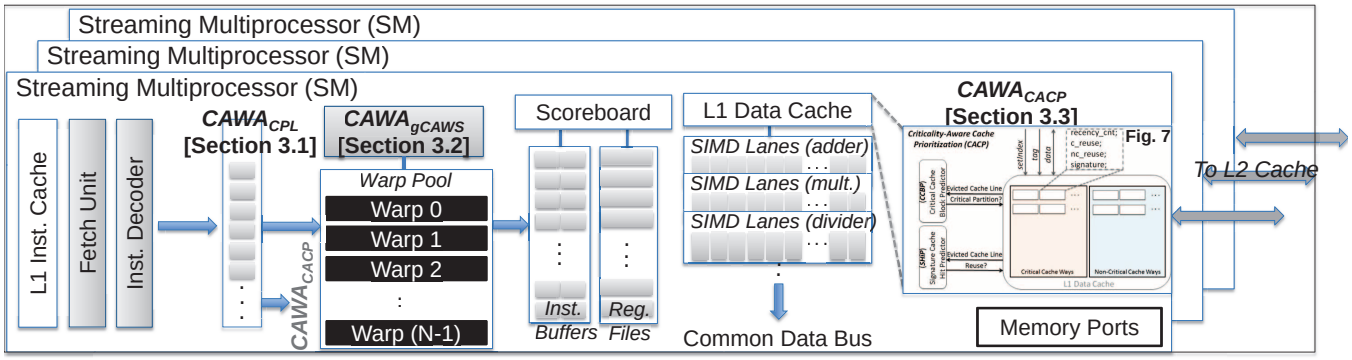

Figure 5: Modern GPU Architecture with CAWA.

```
if(n.hasNotBeenVisited)
    inst-1
    …
    inst-m
else
    inst-1
    …
    inst-n
```

	Branch path	Per-warp #inst.
Thread Branch Divergence: *Yes*		m+n
Thread Branch Divergence: *No*	not-taken	m
	taken	n

Figure 6: The instruction count disparity caused by branch.

scheduling delay imposed onto the critical warps.

In addition to reducing critical warp execution time by allocating more compute resources, CAWA also proactively reserves a certain amount of cache capacity for data that is useful to critical warps ($CAWA_{CACP}$). By doing so, CAWA ensures a certain degree of performance guarantee for the critical warps by reducing the amount of long latency cache misses experienced by the critical warps. Next, we present the three major components in CAWA in detail.

3.1. Critical Warp Identification with Criticality Prediction Logic (CPL)

To identify critical warps at runtime, we develop a criticality prediction logic (CPL) to monitor the execution progress of individual warps in the scheduler's pool by implementing a criticality counter per warp. The per-warp criticality counter represents the execution progress of each warp and is updated based on (1) the degree of instruction count disparity caused by diverging branch behavior, and (2) stall latencies caused by shared resource contention. The per-warp criticality counters are used by our proposed scheduler and cache prioritization schemes for critical warp acceleration.

Dynamic Instruction Count Disparity To consider the influence of workload imbalance and diverging branch behavior on warp criticality, we design CPL to update per-warp criticality counters based on the number of instructions in the executed branch path. Figure 6 shows a simple example to highlight the possibility of diverging dynamic instruction counts per warp based on the branch path behavior. Warps that experience thread-level branch divergence will have to execute *m*+*n* number of instructions while other warps will execute either *m* or *n* instructions based on the branch outcome.

Algorithm 2 An example of the instruction-based CPL.

1: $[PC_1]$ \$L0 : @p0 *bra* \$L2 ▷ jump to PC_4 if p0 is true (ΔInst = PC_4 - PC_1 + 1)
2: $[PC_2]$ \$L1 : *add.u64* %r1, %r1, %1 ▷ jump to PC_5 (ΔInst = PC_5 - PC_1 + 1)
3: $[PC_3]$ *bra* \$L3
4: $[PC_4]$ \$L2 : *sub.u64* %r1, %r1, %1
5: $[PC_5]$ \$L3 : *mov.u64* %r2, %r1, %r2

Depending on the values of *m* and *n*, even without branch divergence, warps could face a significantly different amount of instructions for execution. This could translate to diverging warp execution time.

Based on the branch outcome, CPL updates the per-warp criticality counter accordingly with the inferred size of the basic block determined with the current branch instruction pointer (*currPC*) and the target instruction pointer (*nextPC*). By doing so, CPL would increment or decrement the per-warp criticality counter. In addition to the branch outcomes, CPL also decrements the criticality counter whenever an instruction is committed in order to balance the execution progress.

Algorithm 2 demonstrates an example of how the instruction-based CPL works. If branch divergence occurs at PC_1 for a particular warp, the warp has to run through all three instructions (PC_2, PC_3, and PC_4). On the other hand, there is no branch divergence and depending on the branch outcome, the warp will execute either one (PC_4) or two (PC_2 and PC_3) instructions. After executing PC_1, the target address become available and is used to calculate the additional dynamic instructions per-warp by CPL.

Stall Latency from Shared Resource Contention and Scheduling In addition to updating the per-warp criticality counters based on dynamic execution progress, CPL records additional stall latencies experienced due to shared resource contention as well as scheduler delays, while updating the criticality counter. CPL monitors the stall cycles between the current and the next instruction execution for each warp and increment the criticality counter accordingly for all warps. Algorithm 3 presents the criticality counter update mechanism based on stall cycles in CPL, where the *stallCycle* represents the total stall time between executing two consecutive instructions.

Algorithm 3 The criticality counter with stall cycles.

```
1: function WARPSCHEDULER                ▷ select a ready warp to execute
2:     while NotVisitedAllWarps do       ▷ select warps in the order based on the scheduling policy
3:         w ← findNextWarp()
4:         if w.isRready() then          ▷ stallCycles = total stall time between two consecutive instructions.
5:             w.nStall ← w.nStall + stallCycles
6:             InstructionExecute(w)
7:             break
8:         end if
9:     end while
10: end function
```

Summary for CPL Update The per-warp criticality counter is updated as follows:

$$nCriticality = nInst * w.CPI_{avg} + nStall; \quad (1)$$

where *nCriticality* represents the value of the per-warp criticality counter, *nInst* represents the relative instruction count disparity between the parallel warps, $w.CPI_{avg}$ represents the per-warp average CPI, and *nStall* is the stall cycles incurred by shared resource contention and the scheduler.

3.2. greedy Criticality-Aware Warp Scheduler (gCAWS)

With the critical warp identified by CPL, gCAWS is designed to give more compute resources to critical warps by prioritizing the execution of critical warps over other warps and by providing a larger time slice to warps in a greedy manner. gCAWS combines the strengths of the state-of-the-art schedulers–Greedy-Then-Oldest (GTO) [34] and Criticality-Aware Warp Scheduling (CAWS) [20]. At each cycle, gCAWS selects a ready warp for execution based on the degree of warp criticality determined by the per-warp criticality counter in CPL. If there are multiple warps having the same criticality, the warp scheduler will selects the oldest one based on the GTO algorithm. Then gCAWS greedily executes instructions from the selected critical warp until this particular warp has no further available instructions. Consequently, the critical warp not only receives a higher scheduling priority but also benefits from a larger time slice.

3.3. Criticality-Aware Cache Prioritization (CACP)

In addition to allowing the critical warps to access pipeline resources more often and for a longer time duration, Criticality-Aware Cache Prioritization (CACP) is designed to allocate a certain fixed amount of the L1 data cache capacity to data that will be used by the critical warps for performance guarantee. The insight which CACP builds upon is that *not all cache lines have equal importance*—Data that will be used by critical warps is latency-critical and should be treated with higher priority at the L1 data cache. To do so, among all incoming cache lines, CACP first predicts critical cache lines—cache lines that will be used by critical warps—with the critical

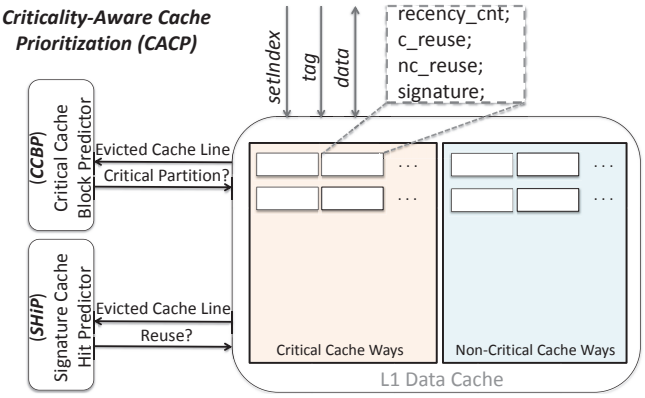

Figure 7: Criticality-Aware Cache Prioritization.

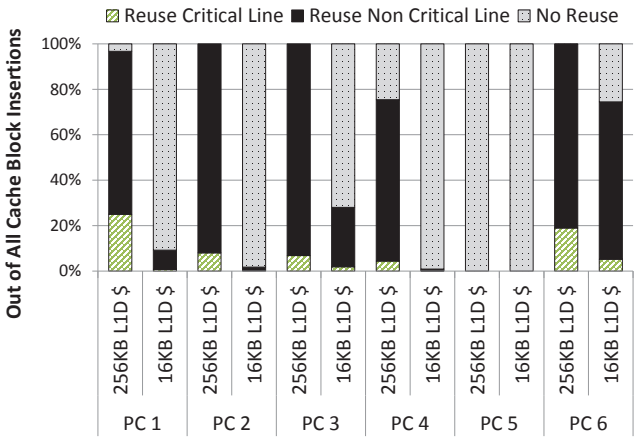

Figure 8: Reuse behavior of different PCs for bfs.

cache block predictor, and retains these critical cache lines in the cache partition reserved for critical warps. Figure 7 shows the proposed CACP scheme and the interface between the different components and the L1 data cache.

Cache Partitioning for Cache Block Criticality CACP partitions the L1 data cache into two parts in the granularity of ways: critical cache ways and non-critical cache ways. The number of ways dedicated to critical versus non-critical cache blocks are determined through experimental analysis of various benchmarks. Through sensitivity analysis, we find that we achieve the best overall performance when 8 out of 16 ways are dedicated to critical cache blocks. A design similar to [31] can be integrated to dynamically tune the size of the critical and non-critical cache partitions based on the run-time needs of an application.

Critical Cache Block Predictor (CCBP) for L1 D-Cache Based on the unique characteristics of GPGPU—relatively small instruction footprint but diverse reuse behavior—we carefully design a Critical Cache Block Predictor (CCBP) to differentiate critical cache lines from non-critical cache lines such that an incoming cache block will either be inserted in the cache partition reserved for critical or non-critical cache lines. Figure 8 shows that there are six memory instructions in this particular `bfs` kernel, which all warps execute. The left bar

within each memory instruction represents the reuse pattern of critical cache blocks in a 256KB data cache whereas the right bar represents the reuse pattern of critical cache blocks in the baseline 16KB data cache. First, when the cache is large enough, cached blocks have a high likelihood of reuse by both critical or non-critical warps. However, the L1 data cache in GPUs are often too small to accommodate the active working set of the entire application. The second observation from Figure 8 is that the reuse patterns for the various memory instructions are different. For instance, the majority of the cache blocks brought by *PC-5* never receive any reuse before being evicted from the cache. With the two key observations, we will design CCBP that learns the reuse patterns for cache blocks based on the insertion instructions and predicts which cache blocks will be reused by critical warps at runtime.

CCBP is built upon the idea of the signature-based cache hit predictor (SHiP) that was originally proposed for the last-level CMP cache [38]. CCBP learns whether an incoming cache line will be used by critical warps or not based on a signature. We design the signature for CCBP to be a combination of instruction program counters (PCs) and memory address regions as the two pieces of information have been shown to be useful in learning and correlating cache block reuse patterns [17, 18, 19, 38]. A signature is formed by xor-ing the lower 8 bits of an instruction PC and the lower 8 bits of the memory address, and is used to index into the CCBP which is a simple array of 2-bit saturating counters. The algorithm for CCBP is outlined in Algorithm 4 in detail.

Modified Signature-based Cache Hit Predictor In addition to CCBP, CACP includes an additional signature-based cache hit predictor (SHiP) that learns and predicts the reuse pattern of any incoming cache blocks based on the same signature used for CCBP. The outcome of SHiP is used to guide the insertion position of the cache block. For example, if a 2-bit re-reference interval prediction replacement policy is used [12], the outcome of SHiP will guide a cache block insertion position to be in the *long* (re-reference prediction value = 2) versus in the *distance* (re-reference prediction value = 3) re-reference prediction.

CACP Summary To protect the critical warp data from getting thrashed in the L1 data cache, we partition the L1 data cache into critical cache partition and non critical cache partition. We capture the limited but important re-references available in the L1 data cache by designing two predictors, CCBP, and SHiP. Both CCBP and SHiP are indexed using signatures formed using PC information augmented with memory address region information.

CCBP plays the crucial role of identifying cache lines that will be reused by critical warps and inserts these cache lines in the critical partition of the cache. To maximize hits, SHiP is used so that only the cache lines that receive re-references are retained in the cache.

With the help of CCBP and SHiP, CACP is able to capture both cache lines that have intra-warp and inter-warp localities. Furthermore, CACP complements the gCAWS warp scheduling scheme by reserving larger cache space to critical warps that are being identified and prioritized by gCAWS. Thus, CAWA is able to take the coordinated approach of criticality prediction, scheduling, and cache prioritization to provide the best performance gain.

Algorithm 4 The CACP pseudo code

```
1: function CACHEFILL(req)  ▷ select partition and insert cache line
2:     line.signature ← (req.pcXORreq.addr)
3:     if CCBP[line.signature] > Threshold then  ▷ predicted to be critical line
4:         L1DCache.CriticalPartition.insert(line)
5:         L1DCache.CriticalPartition.setInsertion Position(line)  ▷ set SHiP insertion position
6:     else  ▷ predicted to be non-critical line
7:         L1DCache.NonCriticalPartition.insert(line)
8:         L1DCache.NonCriticalPartition.setInsertion Position(line)  ▷ set SHiP insertion position
9:     end if
10: end function
11: function CACHEHIT(req)
12:     if InCriticalPartition(line) then
13:         L1DCache.CriticalPartition.setPromotion Position(line)  ▷ set SRRIP promotion position
14:     else
15:         L1DCache.NonCriticalPartition.setPromotion Position(line)  ▷ set SRRIP promotion position
16:     end if
17:     if IsCriticalWarp(req.WarpID) then  ▷ Correct prediction; increment CCBP
18:         line.c_reuse ← true
19:         CCBP[line.signature]++
20:         SHiP[line.signature]++
21:     else  ▷ Hit is from non-critical warp
22:         line.nc_reuse ← true
23:         SHiP[line.signature]++
24:     end if
25: end function
26: function EVICTLINE(line)
27:     if (line.c_reuse = false) AND (line.nc_reuse = true) AND (L1DCache.Partition = CriticalPartition) then  ▷ Incorrect prediction, line should have been inserted in Non-Critical Partition; decrement CCBP
28:         CCBP[line.signature]−−
29:     else if (line.c_reuse = false AND line.nc_reuse = false) then  ▷ No reuse from this signature
30:         SHiP[line.signature]−−
31:     end if
32: end function
```

4. Methodology

To evaluate our coordinated CAWA design, we use GPGPU-sim (version 3.2.0) [1] to explore the behavior of GPGPU applications. GPGPU-sim is a cycle-level performance simulator that models a general-purpose GPU architecture supporting

Architecture	*NVIDIA Fermi GTX480*
Num. of SMs	*15*
Max. # of Warps per SM	*48*
Max. # of Blocks per SM	*8*
# of Schedulers per SM	*2*
# of Registers per SM	*32768*
Shared Memory	*48KB*
L1 Data Cache	*16KB per SM (8-sets/16-ways)*
L1 Inst Cache	*2KB per SM (4-sets/4-ways)*
L2 Cache	*768KB unified cache (64-sets/16-ways/6-banks)*
Min. L2 Access Latency	120 cycles
Min. DRAM Access Latency	220 cycles
Warp Size (SIMD Width)	*32 threads*

Table 1: GPGPU-sim configurations.

Benchmark	Data Set	Category
bfs	65536 nodes	Sens
b+tree	1 million nodes	Sens
heartwall	656x744 grey scale AVI	Sens
kmeans	494020 nodes	Sens
needle	1024x1024 nodes	Sens
srad_1	502x458 nodes	Sens
streamcluster (small)	32x4096 nodes	Sens
backprop	65536 nodes	Non-sens
particle	128x128x10 nodes	Non-sens
pathfinder	100000 nodes	Non-sens
streamcluster (mid)	64x8192 nodes	Non-sens
tpacf	487x100 nodes	Non-sens

Table 2: GPGPU Benchmarks and their data set sizes. Benchmarks are classified into *Sens* and *Non-sens* based on their execution time disparity and sensitivity to L1D cache performance.

NVIDIA CUDA[26] and its PTX ISA[25]. We run GPGPU-sim with the default configuration representing the NVIDIA Fermi GTX480 architecture and configure the per-SM L1 data cache as 16-way set-associative. Table 1 describes the simulation configuration and parameters used for the design evaluation for CAWA.

We select representative GPGPU applications from the Rodinia [5, 6] and Parboil [36] benchmark suites to evaluate the performance improvement of the coordinated CAWA design in GPUs. Table 2 lists the details of the benchmarks and their data sets used to evaluate the CAWA design. Since CAWA mainly aims to improve the performance of those applications with irregular execution behavior as well as cache utilization, we categorize these benchmarks into two sets based on their execution time disparity and sensitivity to L1D cache performance as sensitive (Sens) or non-sensitive (Non-sens).

5. Evaluation Results and Analysis

5.1. Performance Overview

Overall, CAWA improves GPGPU performance by an average of 23% compared to the baseline Round-Robin (RR) warp

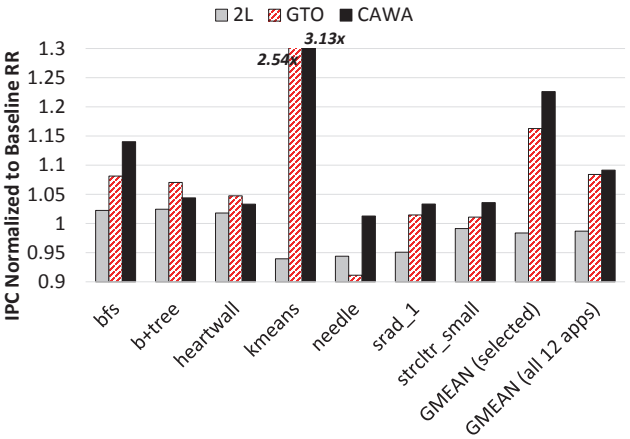

Figure 9: Performance of CAWA design.

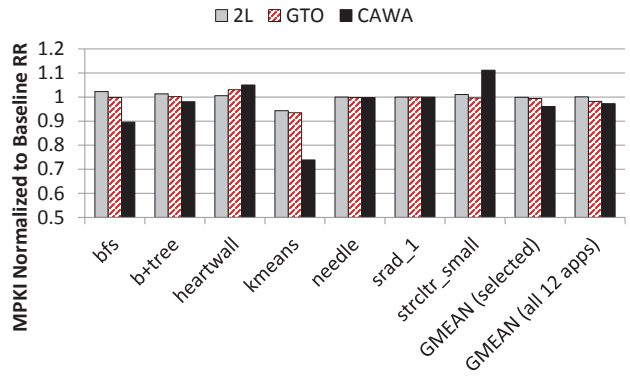

Figure 10: L1D cache MPKI performance of CAWA design.

scheduler for *Sens* applications and by an average of 9.2% over all applications, as Figure 9 shows. In particular, CAWA speeds up the performance of `kmeans` (which suffers from severe cache thrashing) the most, by a significant 3.13 times over the baseline. We also compare the performance of CAWA with two other state-of-the-art warp schedulers, i.e., GTO and the 2-level warp scheduler [24]. Across the seven GPGPU applications that are scheduling policy- and cache sensitive, CAWA improves the performance the most by an average of 23% while GTO and the 2-level scheduler improves the performance by 16% and -2% respectively.

For memory-intensive GPGPU applications, such as `kmeans`, performance is mainly restricted by the efficiency of the data caches. Because of the large amount of data that is streamed over the relatively small L1 data caches, the cached data is evicted from the L1 caches before it receives any additional reuses. GTO alleviates the cache thrashing problem by limiting the number of warps that could be active such that the active working set is kept small and the intra-warp data locality can be better captured in the L1 data cache. The significant performance improvement from CAWA is achieved for a similar reason. The greedy criticality-aware scheduler, $CAWA_{gCAWS}$, is able to limit the number of active warps such that the aggregate working set can well fit into the L1 data

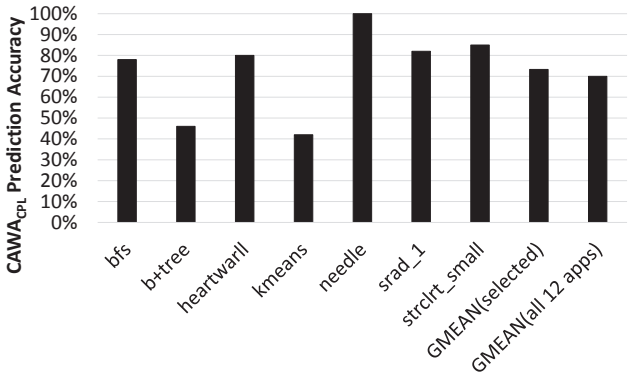

Figure 11: The criticality prediction accuracy of $CAWA_{CPL}$.

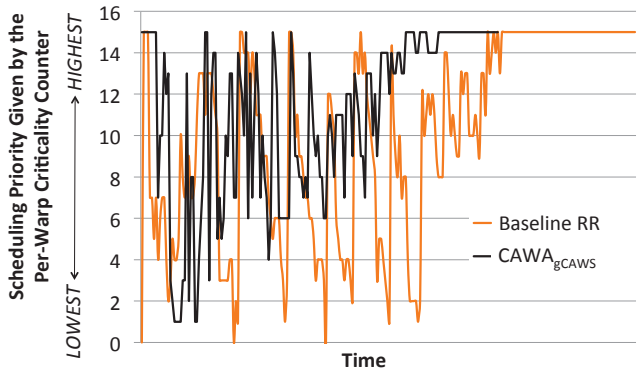

Figure 12: Critical warp's scheduling priority over time of `bfs`.

cache. Furthermore, $CAWA_{CACP}$ reserves a fixed cache capacity for data useful to the critical warps thereby reducing the degree of interference between critical and non-critical cache blocks.

Figure 10 shows the MPKI results for the GPGPU workloads under the 2-level scheduler, GTO, and CAWA, respectively. Overall, CAWA reduces the L1 cache miss rates the most when compared to the other two schemes. For memory-intensive applications, such as `kmeans`, CAWA significantly reduces the cache miss rate by 26.2%. For other applications, such as `heartwall` and `streamcluster-small`, MPKI is increased under CAWA. This is because $CAWA_{CACP}$ prioritizes critical cache blocks over non-critical cache blocks over the baseline cache replacement policy that is designed to minimize cache misses. Although MPKI is increased for `heartwall` and `streamcluster-small`, the corresponding IPC performance is improved by 3.3% and 3.6% respectively. This is because CAWA trades off cache blocks that may be used more often (with better locality) with cache blocks that are critical.

5.2. Performance Analysis for $CAWA_{CPL}$

The critical warp prediction mechanism is a vital component in CAWA and is used to guide compute and memory resource prioritization. To evaluate the accuracy of $CAWA_{CPL}$, we compare the periodic prediction outcomes with the slowest, critical warp based on its total execution time. To calculate the prediction accuracy, we define that if a warp's criticality is larger than 50% of warps in a thread-block, this warp is a slow warp. Since warp criticality could change at runtime which is not captured by the static warp execution time analysis, it is difficult to calculate the prediction accuracy. Thus we count the prediction accuracy as the frequency that the critical warp is identified as a slow warp. Figure 11 shows the warp criticality prediction accuracy comparison. On average, $CAWA_{CPL}$ can accurately identify critical warps as a slow warp with a prediction accuracy of 73%[2]. Since $CAWA_{CPL}$ learns the sources and degree of delay dynamically and reflects the delay and execution progress in the per-warp criticality counters, $CAWA_{CPL}$ is able to adjust its warp criticality prediction outcomes at runtime.

[2]CPL results in an 100% prediction accuracty for `needle` because it is an application which lacks warp-level parallelism, i.e., a thread-block has only one or two warps

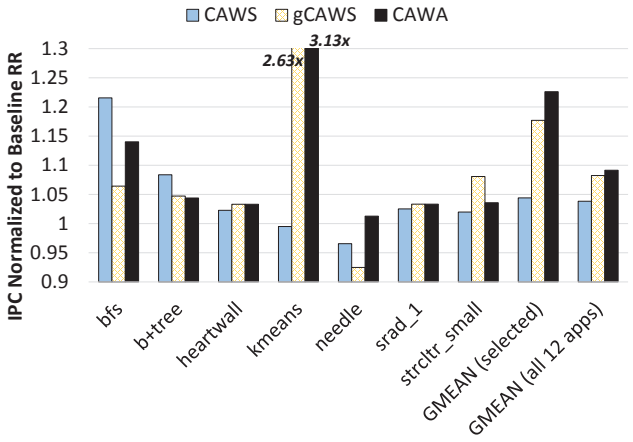

Figure 13: The performance improvement of $CAWA_{gCAWS}$.

5.3. Performance Analysis for $CAWA_{gCAWS}$

At runtime, $CAWA_{gCAWS}$ prioritizes warps based on their criticality updated by $CAWA_{CPL}$. To understand how $CAWA_{gCAWS}$ accelerates warp execution, we investigate the scheduler's decision based on $CAWA_{CPL}$'s critical warp prediction over time. Figure 12 shows the per-warp criticality counter of the static critical warp over time (`bfs`) under the baseline scheduler and under $CAWA_{gCAWS}$. In this particular `bfs` thread block, there are a total of 16 warps; therefore, the critical warp could be ranked in the warp pool with the lowest priority (0 of the y-axis) or with the highest priority (15 of the y-axis) based on $CAWA_{CPL}$'s prediction outcomes. When the critical warp has the highest priority, $CAWA_{gCAWS}$ will schedule it for immediate execution. When compared to a criticality-oblivious scheduler, e.g., the baseline RR, the critical warp is treated equally with all other warps such that it has an equal chance to be selected for execution. In contrast, $CAWA_{gCAWS}$ will proactively schedule the critical warp until its execution progress has improved ($CAWA_{CPL}$ adjusts the criticality counter accordingly). As Figure 12 shows, $CAWA_{gCAWS}$ ranks the critical warp with a higher priority and schedules the critical warp for execution more frequently than the baseline round robin scheduler.

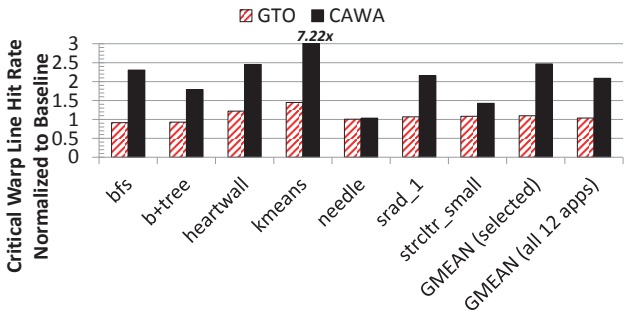

Figure 14: L1D cache critical warp hit rate of CAWA design.

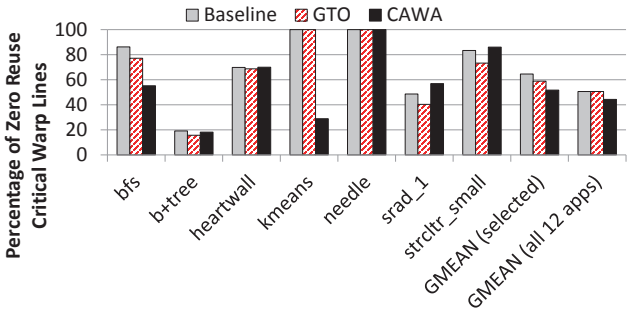

Figure 15: L1D cache zero reuse critical warp lines.

Consequently, with the additional compute resources made available by $CAWA_{gCAWS}$, the execution time of the critical warp is significantly improved.

Figure 13 shows the performance improvement comparison for CAWS [20] with the oracle warp criticality knowledge, $CAWA_{gCAWS}$, and CAWA. With the oracle warp criticality information obtained offline, CAWS performs best on small GPU kernels such as `bfs`, `b+tree` and `needle`. This is because for these small kernels, the prediction and training overhead of $CAWA_{CPL}$ is relatively high. Although $CAWA_{CPL}$ has a good prediction accuracy, $CAWA_{gCAWS}$ and CAWA are not able to achieve the potential speedup under using the oracle knowledge. On the other hand, for large kernel code such as `heartwall` and `srad_1`, our proposed $CAWA_{gCAWS}$ and $CAWA_{CPL}$ can further improve performance compared with CAWS.

We also note that $CAWA_{gCAWS}$ and CAWA achieve a greater performance improvement on `kmeans` than CAWS. This is because `kmeans` has heavy memory contention and prefers to run with fewer number of threads/warps. $CAWA_{gCAWS}$ adopts a greedy scheme to temporarily limit the number of active warps while minimizing warp criticality and resource contention. Because CAWS does not limit the number of active warps to mitigate the memory contention, $CAWA_{gCAWS}$ and CAWA outperforms CAWS on `kmeans`.

Overall CAWA can obtain an additional 5% IPC improvement on selected benchmarks compared with $CAWA_{gCAWS}$. This additional performance improvement is due to the cache prioritization with $CAWA_{CACP}$. However, `b+tree` and `strcltr_small` have a slight performance degradation under CAWA. This is because these two particular applications have high degree of inter-warp data reference and spatial locality. While memory requests from the critical warps have higher priority to be allocated, $CAWA_{CACP}$ does not take the inter-warp reference pattern into account. Therefore, warps may encounter longer memory access latency with $CAWA_{CACP}$.

5.4. Performance Analysis for $CAWA_{CACP}$

Next, we perform analysis for the cache prioritization scheme. CAWA relies on the criticality-aware cache prioritization scheme, $CAWA_{CACP}$, to accelerate the execution of critical warps, thereby reducing latencies coming from the memory subsystem. $CAWA_{CACP}$ separates the data cache to the critical and the non-critical partitions explicitly. Based on the prediction outcome from the Critical Cache Block Predictor and from the Signature-based Cache Hit Predictor, cache blocks are inserted into either the critical or the non-critical cache partitions with the appropriate insertion positions. Figure 14 shows the normalized cache hit rate received by critical warp memory requests under CAWA when compared to the baseline. The explicit cache partitioning with CCBP significantly improves the hit rate for critical warps, by an average of 2.46 times and by as much as 7.22 times for `kmeans`, which outperforms other state-of-the-art warp schedulers. Schedulers, e.g., GTO, that are criticality-oblivious, will not specifically improve the memory performance specifically for critical warps; therefore, the cache hit rate performance is less consistent across the applications.

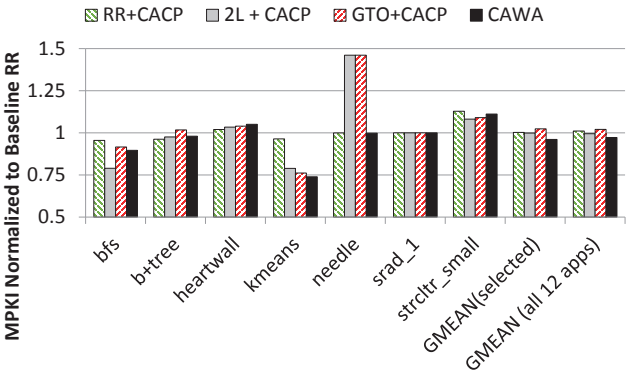

Figure 16: L1D cache MPKI performance of $CAWA_{CACP}$ design with different warp scheduling policies.

To examine the utilization of the cache partition dedicated to critical warps, we compare the number of cache blocks that have never been reused before being evicted from the L1 data cache under the baseline and CAWA. Figure 15 shows that, in the baseline, a majority of cache blocks useful to critical warps, i.e., 44.3%, have never been reused before being evicted from the cache. This is because of the severe interference in the L1 data cache shared between critical and non-critical cache blocks. $CAWA_{CACP}$ is able to reduce such interference by explicitly partitioning and reserving the cache for critical cache blocks; therefore, data retained in the cache is critical and has high locality.

Next, to understand how well $CAWA_{CACP}$ performs in isolation from the scheduler ($CAWA_{gCAWS}$) and in the presence

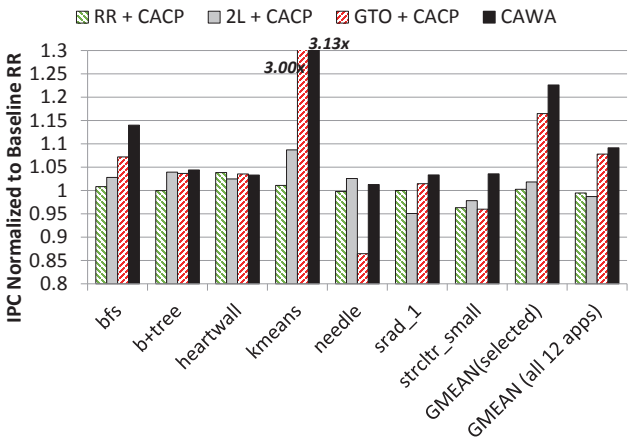

Figure 17: L1D cache IPC performance of $CAWA_{CACP}$ design with different warp scheduling policies.

of other state-of-the-art warp scheduling algorithms, we apply $CAWA_{CACP}$ to the baseline RR, GTO, and the 2-level schedulers. Independent from the warp schedulers, $CAWA_{CACP}$ uses the warp criticality prediction from $CAWA_{CPL}$ for cache prioritization. Figure 16 shows the MPKI reduction for the different warp schedulers under the influence of $CAWA_{CACP}$ and Figure 17 shows the corresponding IPC performance results. When $CAWA_{CACP}$ is used in conjunction with the various state-of-the-art schedulers, additional performance improvement is achieved from 2% to 16.5% while our proposed coordinated management still performs the best.

5.5. Discussion for Warp Criticality

5.5.1. Criticality in the Context of a Scheduler There have been warp scheduling algorithms proposed to alleviate resource contention by greedily executing a small group of warps with a higher priority, e.g., [24, 34]. In this type of unfair warp schedulers, warps are scheduled with different priorities based on the cache utilization and warp cache sharing behavior. Such schedulers could intentionally create warp criticality, depending on the scheduling order of warps in a thread block, and thus the warp criticality inherited from applications can be skewed, while the concept of warp criticality proposed in this paper is most directly applicable to fair warp schedulers.

5.5.2. Applying Criticality to Existing Schedulers The idea of warp criticality is orthogonal to the warp scheduler's design, particularly for applications whose warp criticality stems from software code implementation (workload imbalance/branch divergence). Different schedulers may experience different degrees of warp criticality since criticality is often caused by multiple sources as described in Section 2. The notion of warp criticality can also be integrated into existing, advanced schedulers with careful design modification to further improve GPU performance by eliminating warp execution time disparity, leading to improved pipeline utilization. As this paper shows, $CAWA_{gCAWS}$ applies CPL on top of GTO [34] and improves performance by 7% for the selected, scheduling and cache sensitive applications.

6. Related Work

Prior work have looked at various issues that limit GPU performance, including warp scheduling policies and memory subsystem design. In this section, we discuss prior works that are most related to this paper.

6.1. Thread/Warp Criticality Prediction

The thread criticality problem on CMP has been addressed in many works. Bhattacharjee and Martonosi [2] observed that the thread criticality problem for parallel threads executing on CMPs. Since the performance of a parallel application is constrained by the critical (the slowest running) thread, the execution time of the parallel application is significantly affected by the execution time of the critical thread. Bhattacharjee and Martonosi designed the thread criticality predictor (TCP) to identify the critical thread based on per-thread cache behavior. TCP is then used to guide the task stealing as well as dynamic voltage and frequency scaling (DVFS) of CMPs. Inspired by the thread criticality concept, Ebrahimi et al. exploited the resource contention at a spin lock as the metric to evaluate the critical thread [8], whereas Bois et al. proposed a stack based approach to measure thread criticality by monitoring the number of waiting threads in a certain interval [3]. Although the concept of thread criticality in CPU parallel applications is similar to warp criticality in GPGPU workloads, due to the distinct difference between CPU and GPU architectures, the effects introduced by the critical thread(s) and warp(s) vary as well. Because of the fast context-switching and massive multi-threading in modern GPUs, dynamic warp criticality prediction is challenging. Lee and Wu [20] were the first to show that the performance of modern GPU architectures is also strictly constrained by the performance of critical warps. However, there still lacks a robust approach to dynamically identify the critical thread/warp on GPU platforms, which is the focus of the work performed in this paper.

6.2. GPU Warp Scheduling

Many recent works [11, 15, 16, 24, 34, 35] focused on improving the memory aspect of GPUs by modifying warp scheduling algorithms. Gebhart et al. and Narasiman et al. designed a 2-level scheduler to split warps into different subgroups and keep only one group of warps active at a time [11, 24]. The warp scheduler is only able to issue instructions from the active subgroup of warps, so that the resource contention problem can be alleviated. Jog et al. further improved the 2-level scheduler by splitting warps with continuous IDs to different subgroups. Because memory requests from continuous warps have high probability to fall into the same L2 cache/DRAM bank, resulting in bank conflicts. With this warp grouping algorithm, the GPU performance can be improved by avoiding bank conflicts at the L2 cache and DRAM. Rogers et al. proposed a warp scheduler design by monitoring memory contention [34]. The warp scheduler dynamically tunes the number of active warps based on the degree of data cache contention. If the cache con-

troller detects cache lines in the L1 data cache are frequently evicted due to the interference from inter-warp accesses, the warp scheduler will limit the number of active warps. These algorithms prevent all warps from stalling at the same time by ensuring the warps in the pool do not encounter long latency memory operations or compete for register file banks at the same time. These scheduling policies allow the GPU warp scheduler to tolerate memory latency better while reducing the idle time of GPU cores. In contrast, the gCAWS design in this paper aims to resolve resource contention by designing a novel warp scheduling order based on warp criticality and by allowing critical warps to execute with larger time slices.

6.3. Branch Divergence

Many warp scheduling algorithms [9, 10, 23, 24, 35, 37] have been proposed to alleviate the effects of branch and memory divergence on GPUs. When branch or memory divergence happens, the utilization of GPU resources becomes sub-optimal. A common algorithm to reduce the degree of branch divergence is dynamically re-packing threads into different warps at runtime [9, 10, 23, 37], so that threads having the same execution path can be executed together. Instead of re-forming warps, Rhu and Erez [32] proposed an interleaving execution technique to overlap a divergent warp's execution. This line of prior works target to increase the pipeline utilization of functional units instead of alleviating the warp criticality problem which our proposed CAWA aims to tackle. With the previously proposed designs, the branch diverging behavior will be improved. However, the diverging warp execution time still persists, which will benefit from the CAWA design presented in this paper.

6.4. GPU Memory Subsystem and Resources Contention

Studies relevant to CMP cache designs mainly target to maximize overall cache hit rates [12, 17, 18, 29, 30, 38]. However, the architecture of modern GPUs is massively-parallel when compared to its CMP counterpart. The per-thread cache capacity in GPUs is relatively small, resulting in serious data thrashing problems [12, 29, 30]. Recent works, e.g, [7, 33], looked at how well cache insertion/replacement policies proposed for CMP caches perform in the massively-parallel GPU environment and found that directly employing state-of-the-art cache management policies for CMPs does not effectively alleviate the data thrashing problem in the GPU caches.

Instead of aiming to improve cache hit rates, many works, e.g. [7, 13, 14, 39], investigated techniques to improve GPU's memory subsystems by cache bypassing, which can significantly reduce the demand of memory/bus bandwidth. Jia et al. [13] and Xie et al. [39] proposed using compilers to perform off-line analysis to find out the load/store instructions or memory regions which may have poor spatial locality. GPUs take these hints generated by compilers to bypass data cache in order to mitigate the bus bandwidth contention. On the other hand, Jia et al. [14] designed a memory request prioritization buffer (MRPB) which reorders the memory requests and/or bypasses the requests from the L1 data caches. With MRPB, requests coming from the same warp or thread-block are serviced together to maximize locality. MRPB is able to effectively improve data cache locality and alleviate memory contention. However, all of these schemes do not take criticality into account. The critical warp may wait for longer time to be served. In addition to the previously proposed cache management policies designed for GPUs, this paper presents a cache prioritization scheme to improve the cache performance and tailors the design for critical warp acceleration.

7. Conclusion

This work introduces a new coordinated compute and memory resource prioritization design for GPGPU critical warp acceleration. We quantify the sources of the warp criticality problem in the massively-parallel GPU environment.

Built upon the insights observed from the warp criticality characterization results, this work proposes CAWA to dynamically predict critical warps and accelerate the execution of the critical warps with a higher scheduling priority and with a larger scheduling time slice. Furthermore, CAWA reserves a partition of the L1 data cache for data predicted-to-be-useful for the critical warps; therefore, the interference between critical and non-critical cache blocks is minimized. Our simulation results shows that the proposed design improves performance by an average of 23% on highly scheduler-sensitive and cache-sensitive workloads.

This work demonstrates that coordinated management of warp scheduler and cache prioritization is necessary for a holistic GPGPU performance improvement. With the knowledge of warp criticality learned and predicted at runtime, warp schedulers and cache insertion/replacement techniques can be designed more intelligently to extract additional performance improvement from modern GPUs. We hope the detailed characterization presented in this paper can offer new insights into the important warp criticality problem and motivate novel solutions for future GPGPU architectures.

Acknowledgements

We would like to thank Wenhao Jia (Qualcomm) and the anonymous reviewers for their feedback. The work was supported, in part, by the Science Foundation Arizona under the Bisgrove Early Career Scholarship and by Arizona State University. The opinions, findings and conclusion or recommendations expressed in this manuscript are those of the authors and do not necessarily reflect the views of the Science Foundation Arizona and ASU.

References

[1] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. of the 2009 IEEE International Symposium on Analysis of Systems and Software (ISPASS'09)*, Boston, MA, USA, April 2009.

[2] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proc. of the 36th IEEE/ACM International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, USA, June 2009.

[3] K. D. Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proc. of the 40th IEEE/ACM International*

Symposium on Computer Architecture (ISCA'13), Tel Aviv, Israel, June 2013.

[4] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. of the 2012 IEEE International Symposium on Workload Characterization (IISWC'12)*, San Diego, CA, USA, November 2012.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, Austin, TX, USA, October 2009.

[6] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proc. of the 2010 IEEE International Symposium on Workload Characterization (IISWC'10)*, Atlanta, GA, USA, December 2010.

[7] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, and W. mei Hwu, "Adaptive cache management for energy-efficient GPU computing," in *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, Cambridge, UK, December 2014.

[8] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proc. of the 44th International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, December 2011.

[9] W. L. W. Fung and T. M. Aamodt, "Thread block compaction for efficient SIMT control flow," in *Proc. of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, San Antonio, TX, USA, February 2011.

[10] W. L. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. of the 37th IEEE/ACM International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010.

[11] M. Gebhart, R. D. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindoholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proc. of the 38th IEEE/ACM International Symposium on Computer Architecture (ISCA'11)*, San Jose, CA, USA, June 2011.

[12] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. of the 37th IEEE/ACM International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010.

[13] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proc. of the 20th ACM International Conference on Supercomputing (ICS'12)*, Venice, Italy, June 2012.

[14] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: memory request prioritization for massively parallel processors," in *Proc. of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*, Orlando, FL, USA, February 2014.

[15] 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 *Proc. of the 40th IEEE/ACM International Symposium on Computer Architecture (ISCA'13)*, Tel-Aviv, Isreal, June 2013.

[16] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," in *Proc. of the 18th IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, Houston, TX, USA, March 2013.

[17] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *Proc. of the 25th IEEE International Conference on Computer Design (ICCD'07)*, Lake Tahoe, CA, USA, October 2007.

[18] S. Khan, Y. Tian, and D. Jimenez, "Sampling dead block prediction for last-level caches," in *Proc. of the 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*, Atlanta, GA, USA, December 2010.

[19] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proc. of the 28th IEEE/ACM International Symposium on Computer Architecture (ISCA'01)*, 2001.

[20] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware warp scheduling for GPGPU workloads," in *Proc. of the 23rd IEEE/ACM International Conference on Parallel Architectures and Compilation (PACT'14)*, Edmonton, AB, Canada, August 2014.

[21] S.-Y. Lee and C.-J. Wu, "Characterizing the latency hiding ability of GPUs," in *Proc. of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14) as Poster Abstract*, Monterey, CA, USA, March 2014.

[22] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, March 2008.

[23] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proc. of the 37th IEEE/ACM International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010.

[24] 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 *Proc. of the 44th International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, December 2011.

[25] NVIDIA, "PTX ISA," 2009. Available: http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf

[26] NVIDIA, "NVIDIA CUDA C programming guide v4.2," 2012. Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation

[27] NVIDIA, "NVIDIA GeForce GTX 980: Featuring Maxwell, the most advanced GPU ever made," September 2014.

[28] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," in *Proc. of the 2014 IEEE International Symposium on Workload Characterization (IISWC'14)*, Raleigh, NC, USA, October 2014.

[29] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of the 34th IEEE/ACM International Symposium on Computer Architecture (ISCA'07)*, San Diego, CA, USA, June 2007.

[30] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer, "Set-dueling-controlled adaptive insertion for high-performance caching," *IEEE Micro*, vol. 28, no. 1, pp. 91–98, January 2008.

[31] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. of the 39th IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Orlando, FL, USA, December 2006.

[32] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," in *Proc. of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*, Shenzhen, China, February 2013.

[33] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architecture," in *Proc. of the 46th International Symposium on Microarchitecture (MICRO'13)*, Davis, CA, USA, December 2013.

[34] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*, Vancouver, BC, Canada, December 2012.

[35] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proc. of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*, Davis, CA, USA, December 2013.

[36] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "The Parboil technical report," in *IMPACT Technical Report (IMPACT-12-01), University of Illinois Urbana-Champaign*, Champaign, IL, USA, March 2012.

[37] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi, "SIMD divergence optimization through intra-warp compaction," in *Proc. of the IEEE/ACM 40th International Symposium on Computer Architecture (ISCA'13)*, Tel Aviv, Israel, June 2011.

[38] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "SHiP: signature-based hit predictor for high performance caching," in *Proc. of the 44th IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, December 2011.

[39] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proc. of the 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*, San Jose, CA, USA, November 2013.