OAWS: Memory Occlusion Aware Warp Scheduling

Bin Wang
Auburn University
bwang@auburn.edu

Yue Zhu
Florida State University
yzhu@cs.fsu.edu

Weikuan Yu
Florida State University
yuw@cs.fsu.edu

ABSTRACT

We have closely examined GPU resource utilization when executing memory-intensive benchmarks. Our detailed analysis of GPU global memory accesses reveals that divergent loads can lead to the occlusion of Load-Store units, resulting in quick consumption of MSHR entries. Such memory occlusion prevents other ready memory instructions from accessing L1 data cache, eventually stalling warp schedulers and degrading the overall performance. We have designed memory Occlusion Aware Warp Scheduling (OAWS) that can dynamically predict the demand of MSHR entries of divergent memory instructions, and maximize the number of concurrent warps such that their aggregate MSHR consumptions are within the MSHR capacity. Our dynamic OAWS policy can prevent memory occlusions and effectively leverage more MSHR entries for better IPC performance for GPU. Experimental results show that the static and dynamic versions of OAWS achieve 36.7% and 73.1% performance improvement, compared to the baseline GTO scheduling. Particularly, dynamic OAWS outperforms MASCAR, CCWS, and SWL-Best by 70.1%, 57.8%, and 11.4%, respectively.

1 Introduction

GPU applications have a wide range of memory access patterns, many of which are very irregular. Despite the high-bandwidth GPU global memory, irregular patterns can stall the GPU memory and degrade the effectiveness of massive parallelism. GPUs have employed a hierarchy of data caches to reduce memory latency and save the on-chip network and off-chip memory bandwidth when there is locality within the accesses. However, the cache is frequently thrashed by divergent memory accesses. To tackle this problem, GPU L1D bypassing has been well studied to alleviate cache contention [1, 2, 3, 4, 5, 6, 7, 8]. For example, MRPB [1] aggressively bypasses L1D whenever an associativity stall occurs, but the cache is still underutilized. Two recent works [9, 10] reported that throttling the number of concurrent warps reduces the accumulated working set so that the contention on cache capacity is alleviated and locality is preserved.

While the aforementioned efforts improved the memory performance of GPU applications, they overlooked some hazardous situations that are faced by GPU memory instructions. In this paper, we have closely examined execution stalls caused by divergent memory accesses in data-intensive GPU benchmarks. Our detailed analysis of GPU global memory accesses reveals that divergent accesses can lead to the occlusion of Load-Store units and a quick depletion of MSHR (Miss Status Holding Registers) [11] entries. This memory occlusion in turn stalls the execution pipeline, degrading the overall performance. Our analysis shows that memory occlusion can significantly delay both coherent and divergent GPU instructions, propagate into more stalls in the pipeline, and deteriorate the overall utilization of GPU. Furthermore, such memory occlusion cannot be mitigated by a simple provision of more MSHR entries, when there is a lack of appropriate balance between warp parallelism and L1D locality.

Turner [12] investigated instruction replay for mitigating the performance impact of pipeline stalls caused by GPU memory operations, aiming to reduce wasteful replays by predicting resource demands of GPU memory operations. In this paper, we propose memory Occlusion Aware Warp Scheduling (OAWS) to monitor the usage of MSHR entries, predict the MSHR requirement of memory instructions, and schedule the warps that can be satisfied by the available MSHR entries, thereby preventing memory occlusion and increasing the effective parallelism among GPU warps. OAWS is designed with both static and dynamic methods to predict the required MSHR entries from GPU warps. Static OAWS predicts the misses from all warps with a fixed miss rate while dynamic OAWS takes into account the varying access patterns of different warps to predict cache misses on a per-warp basis. Particularly, dynamic OAWS has seamlessly integrated the warp priority and a concurrency estimation model in its prediction policy. It effectively prevents memory occlusion by dynamically predicting an optimal number of concurrent warps, and uses more MSHR entries effectively without thrashing L1D.

We have leveraged a wide variety of memory-intensive benchmarks to evaluate the performance of OAWS and demonstrated that OAWS outperforms three state-of-the-art warp scheduling techniques, i.e., Cache-Conscious Wavefront Scheduling (CCWS) [9], Static Wavefront Scheduling (SWL) [9] and Memory Aware Scheduling and Cache Access Re-execution (MASCAR) [13]. Specifically, our experiments show that our static and dynamic versions of OAWS achieve 36.7% and 73.1% compared to the baseline GTO warp scheduling. In addition, dynamic OAWS helps GPU leverage the benefits of more MSHR entries. Compared to GTO, it improves the IPC performance by 73.1%, 89.1%, 92.6%, and 99.4% when there are 32, 48, 64, and 96 MSHR entries, respectively.

The rest of this paper is organized as follows. Section 2 describes the background on GPU architecture and the execution of memory instructions. Section 3 provides an analysis of memory occlusion. Section 4 details the proposed memory occlusion aware

PACT '16, September 11-15, 2016, Haifa, Israel

DOI: http://dx.doi.org/10.1145/2967938.2967947

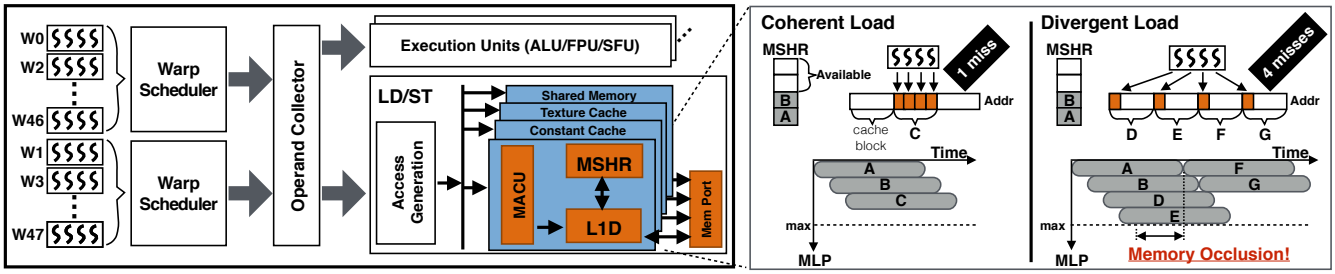

Figure 1: Baseline Streaming Multiprocessor (SM). Other stages of the pipeline are omitted. The MSHR of L1D has 4 entries, two of which have been allocated to track the outstanding memory requests of cache blocks A and B. The remaining MSHR entries are sufficient for a coherent load, but occlude the divergent load so that demand requests of blocks F and G are replayed.

warp scheduling algorithms. The experimental methodology and results are presented in Section 5. Section 6 summarizes the related work, followed by Section 7 that concludes the paper.

2 Background

2.1 GPU and Warp Scheduling

We study a Fermi-like GPU from NVIDIA, which has 32 threads per warp and 32 MSHR entries per Streaming Multiprocessor (SM) as the default configuration. Inside each SM as shown in Figure 1, two warp schedulers independently manage warps with even and odd identifiers [14]. In each cycle, both warp schedulers pick one ready warp and issue its instruction into the SIMD pipeline backend [15, 14, 16], which in our baseline SM mainly consists of *Operand Collector* (i.e., registers), *Execution Units (ALU/FPU/SFU)*, and *Load-Store (LD/ST)* units. To determine the readiness of an instruction, a ready bit is used to track its dependency on other instructions. It is updated in the scoreboard by comparing its source and destination registers with other in-flight instructions [17]. Instructions are ready for scheduling when the ready bits are set, i.e., dependencies are cleared. GPU scheduling logic consists of two stages: qualification and prioritization [18]. In the qualification stage, ready warps are selected based on the ready bit of each instruction. In the prioritization stage, ready warps are prioritized for execution based on a chosen metric, such as cycle-based round-robin [18, 19, 20], warp age [9, 10], instruction age [18, 21], or others that can maximize resource utilization [18]. For example, the Greedy-Then-Oldest (GTO) [9, 10] scheduler maintains the highest priority for an active warp until it is stalled. It then prioritizes the oldest among all ready warps for scheduling.

2.2 GPU Memory Access and Occlusion

According to data destination, GPU memory requests are sent to data cache (L1D), constant cache, texture cache, and shared memory, respectively. Inside GPU LD/ST units, memory accesses are generated by the *Access Generation* unit. A memory instruction is coherent when its memory accesses can be coalesced into 1-2 cache lines; otherwise, it is referred to as a divergent instruction.

For memory instructions to local/global memory, per-thread memory accesses are coalesced into fewer cache lines by the *Memory Access Coalescing Unit (MACU)*. Coalesced accesses are sent to L1D via a single 128-byte port [21]. For a load access, on cache hit, the requested data is loaded to the register file; on cache miss, one demand request is generated to fetch data from the lower memory hierarchy. The Missing Status Holding Register (MSHR) is used to track in-flight memory requests and merge duplicate requests to the same cache line. After MSHR allocation, a memory request is buffered into the *Memory Port* for transfer. An MSHR entry is deallocated after its corresponding memory request is back and all accesses to that block are serviced. L1D does not support coherence, so it evicts cache blocks on stores to global memory. Stores require no MSHR and are directly buffered into the memory port. Memory requests buffered in the memory port are drained by the on-chip network in each cycle when lower memory hierarchy is not saturated.

Memory Occlusion: MSHR is often implemented as a fully-associative structure and thus is limited by capacity. This leads to a structural hazard due to the mismatch between limited memory-level parallelism (MLP) and massive thread-level parallelism (TLP). This hazard can be exacerbated by bursty accesses from divergent loads, causing the propagation of stalls [12] in the execution pipeline. As shown in the right part of Figure 1, the MSHR of L1D has 4 entries. The coherent load with 1 cache miss can be immediately serviced, while the four un-coalesceable accesses of the divergent load suffer from insufficient MSHR entries because only two MSHR entries are available. In this example, the access to block F that misses in L1D is replayed until the memory request of block A is back and its MSHR entry is deallocated. During the access replay, the currently prioritized memory instruction can not make progress, occluding LD/ST units. This occlusion in turn prevents other memory instructions from accessing LD/ST units. We refer to such a scenario as ***Memory Occlusion***. Memory Occlusion degrades memory instruction throughput in LD/ST units and prevents other memory instructions that do not need an MSHR from accessing the L1D.

3 Characterization of Memory Occlusion

We investigate the impacts of memory occlusion by analyzing the breakdown of LD/ST stall cycles, quantifying the global memory access time, and examining the trend of MSHR consumption under the GTO warp scheduling.

3.1 Stalls in LD/ST and Warp Schedulers

When LD/ST units are stalled, a ready memory instruction can not be issued. We refer to such stall cycles as LD/ST stalls. Besides MSHR unavailability and memory port congestion, sequentially processing un-coalesceable memory accesses makes the LD/ST units unavailable to warp schedulers and delays other ready memory instructions accessing L1D. According to these three causes that can stall LD/ST units, we breakdown the LD/ST stall cycles into three categories in Figure 2: 1) coalescing stalls (*LDST_COAL*) — when L1D has successfully serviced one un-coalescable memory access and LD/ST units are stalled by divergent memory accesses; 2) MSHR stalls (*LDST_MSHR*) — when a cache miss can not be processed due to MSHR unavailability; and 3) ICNT stalls (*LDST_ICNT*) — when a cache miss can not be processed due to on-chip network congestion, but MSHR entries are available. Among the memory coherent benchmarks, 2MM, 3MM, SRAD1, SRAD2, and LBM experience a large percentage of *LDST_ICNT*

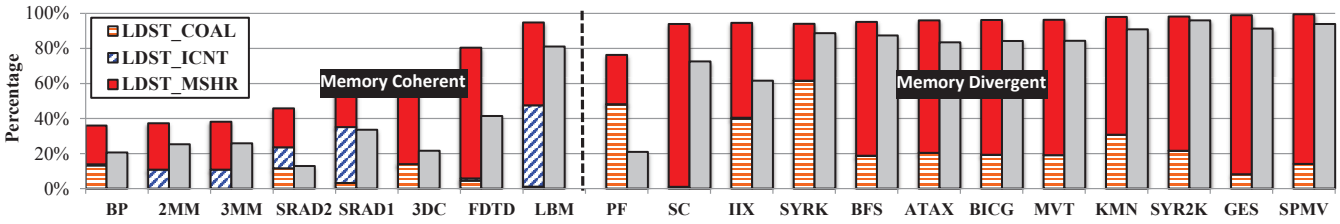

Figure 2: The breakdown of LD/ST stall cycles (the stacked bar on left) and the percentage of cycles for which the warp schedulers are stalled due to LD/ST stalls (the right bar). The dotted line divides all benchmarks into memory coherent (left) and memory divergent (right) ones. Benchmark characteristics and simulator details are summarized in Section 5.

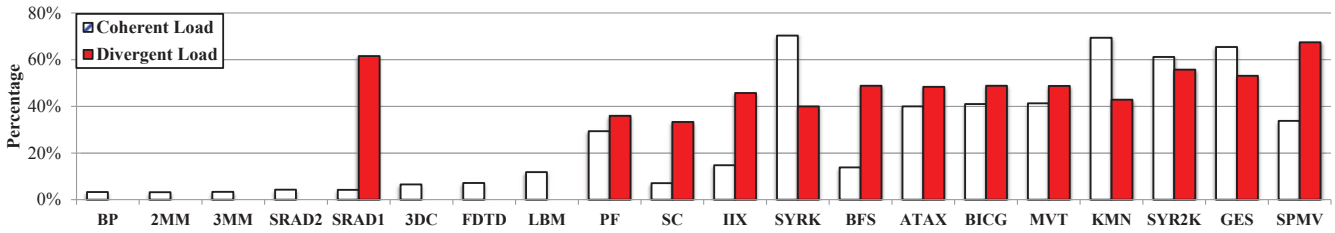

Figure 3: Memory occlusion time in the memory access latency for coherent and divergent loads.

stalls. These five benchmarks write significant amounts of data into global memory, congesting the network from SM to L2 cache. For memory divergent benchmarks, *LDST_MSHR* dominates LD/ST stall cycles, with an average of 66% of total cycles waiting for MSHR entries. The divergent benchmarks are read-intensive. However, read requests impose limited pressure on the network from SM to L2 cache. So *LDST_ICNT* plays a negligible role in these benchmarks.

Although LD/ST units are stalled, warp schedulers can still issue computation instructions into execution units to overlap the stalls in LD/ST units. The capability of overlapping LD/ST stalls explains why warp schedulers (the gray bar) are stalled less than LD/ST units across all of the benchmarks in Figure 2. However, LD/ST stalls eventually deplete ready warps for the schedulers when all warps are waiting to be scheduled to issue memory instructions. For memory coherent benchmarks, on average, warp schedulers waste 28% of the total cycles waiting for the availability of LD/ST units. This percentage increases to 75% for memory divergent benchmarks. Such high stalls in warp schedulers directly lead to severe degradation of instruction throughput.

3.2 Quantifying Memory Occlusion Time

Upon memory occlusion, L1D access latency can be divided into memory occlusion time and L1D hit/miss time. Figure 3 shows the percentage of memory occlusion time in the average L1D access latency for both coherent and divergent load instructions in memory intensive benchmarks. On average, memory occlusion delays account for 4% of L1D access time for all memory coherent benchmarks. SRAD1 has very few divergent loads through its 502 kernel invocations. In memory divergent benchmarks, memory occlusion delays reach 33% and 47% for coherent and divergent loads, respectively. These large delays dramatically prolong memory access time, demanding a higher degree of computation-memory overlap. However, these memory-divergent benchmarks often lack sufficient computation instructions to overlap with memory accesses, causing the warp scheduler stalls shown in Figure 2. It is clear that memory occlusion is a performance destructor for divergent GPGPU workloads, which is the focus in the paper.

Since the lack of MSHR entries is inherently associated with memory occlusion, an intuitive solution is to employ more as a simple solution. We have investigated the impact of more MSHR entries to memory occlusion under different scheduling policies. Our results have shown that, while deeply associated with the number of MSHR entries, memory occlusion cannot be solved by simply adding more MSHR entries (see Section 5.5 for more details).

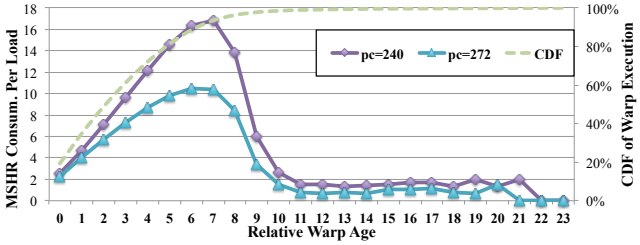

Figure 4: The MSHR consumption of each load instruction in BFS kernel and the CDF of warp execution times when using GTO scheduler. *pc=240* and *pc=272* are the two divergent loads. Age 0 represents the oldest warp.

3.3 Predictability of MSHR consumption

Since the consumption of MSHR entries is a good indicator of potential memory occlusion, we use one kernel from BFS to study MSHR consumption when using a GTO warp scheduler. The two divergent load instructions (pc=240 and pc=272) in this kernel carry high intra-warp locality [9, 10]. Because older warps have higher scheduling priorities under GTO scheduling, older warps are more frequently scheduled and their data blocks have shorter re-reference intervals, leading to lower cache misses and consequently lower MSHR consumption. As shown in Figure 4, the MSHR consumption levels of the two divergent loads are linear to each warp's relative age. The CDF of warp execution times shows that the oldest 8 warps (age=0-7) occupy 94% of total cycles. Note that the warps with even younger ages (age>7) have lower MSHR consumption because they are rarely scheduled, until older warps have very few active threads remained or retired completely. This observation also applies to the other memory divergent benchmarks we have evaluated. Thus, we argue that, because of the existence of a strong linear relationship, the MSHR consumption of a load instruction

can be predicted based on its hit/miss history and its warp's GTO scheduling priority.

4 OAWS: Occlusion Aware Warp Scheduling

We propose memory Occlusion Aware Warp Scheduling (OAWS) to monitor the use of MSHR entries and predict memory occlusion, thereby scheduling warps appropriately to alleviate its impact.

4.1 Main Idea of OAWS

Figure 5 provides an example on the benefit of OAWS. For illustration purpose, the number of MSHR entries in Figure 5 is 8. Assume that the example kernel first fetches data from memory and then computes it. There are 3 ready warps, A, B, and C, to be scheduled (Figure 5(a)) . They are ordered by their arrival times. The memory load instructions in warps A, B, and C are divergent and produce 6, 6, and 2 memory load requests to L2 cache, respectively. The varying number of memory requests for warps are caused by either branch divergence or warp locality variation [9, 10]. Unless otherwise noted, we employ by default the GTO [9] warp scheduler.

Figure 5(b) shows how the GTO scheduler works. The load instructions from warps A, B, and C are issued at T_0, T_1, and T_4, respectively, according to their age. At the time T_2, two MSHR entries are allocated to memory requests from warp B. At this point, MSHR entries are used up and memory occlusion occurs. Therefore, the remaining requests of warp B are continuously replayed until the responses for warp A's memory loads arrive at T_3. At T_5, all data required by warp A are returned from L2 and warp A is ready to be issued for computation. While warp A finishes computation at T_6, warps B and C are stalled due to outstanding memory requests. The computation for warps B and C starts at T_7 and T_8, respectively. As shown in Figure 5(b), GTO causes idle cycles between T_6 and T_7.

In contrast, in Figure 5(c), OAWS predicts that available MSHR entries cannot satisfy warp B's demand, so it prevents warp B from issuing its memory instruction at $T_1^{'}$. OAWS then avoids the memory occlusion that may occur when warp B is scheduled. When warp A's requests return from memory, their MSHR entries are released and warp B is then scheduled. Since the ALU keeps idle, warp A and C are immediately scheduled for computation at $T_4^{'}$ and at $T_5^{'}$, respectively. Finally, warp B is scheduled for computation at $T_6^{'}$. Since warp C is scheduled ahead of warp B, its computation overlaps with warp B's memory load, reducing the time to complete these warps. Although this example shows only three warps and a miss penalty of 12 cycles for simplicity, the benefit of occlusion-awareness could be more significant because the off-chip memory latency is often between 400 and 500 cycles.

4.1.1 Qualification Metric of OAWS

OAWS aims to prevent memory occlusion with a simple logic, i.e., ensuring that enough MSHR entries are available for a divergent load. To this end, it needs to predict the number of cache misses for an incoming divergent load, and take into account the needs of load instructions that have been issued. Thus, the qualification logic of OAWS is to qualify a memory instruction if its predicted cache misses can satisfy the following condition:

$$Miss_{pred}(pc,w) \leq Avail_{mshr} - Miss_{inflight}, \quad (1)$$

where $Avail_{mshr}$ is the number of available MSHR entries for a L1D, $Miss_{pred}(pc,w)$ is the predicted number of cache misses that warp w is going to incur for the memory instruction at address pc, and $Miss_{\text{inflight}}$ is the number of predicted cache misses from in-flight load instructions. At runtime, $Miss_{\text{inflight}}$ is updated as memory instructions are issued or completed by the LD/ST units. Note that stores do not consume MSHR entries because coherent loads rarely consume 2 entries. Coherent loads are predicted to consume one MSHR entry with 1 cache miss. In OAWS, we focus on divergent load instructions which are more likely to cause memory occlusion.

4.1.2 Designing Scheduling Policies for OAWS

To accurately predict the number of cache misses for a divergent load instruction, we have explored both static and dynamic policies for OAWS. Figure 6 presents the microarchitecture for both versions. OAWS is implemented as an extension of the warp scheduler's qualification logic. Conventional qualification logic is used to pick warps that are ready for execution, which is denoted as a N-bit vector *Ready[1:N]* (❶). OAWS relies on the *Divergent Load Classifier (DLC)* to predict $Miss_{\text{pred}}(pc)$ and then re-qualifies ready warps using the logic in Equation 1. The output of OAWS is another N-bit vector *Occlude[1:N]* (❸), in which a bit value 0 denotes a warp predicted to not occlude MSHR entries, 1 otherwise. OAWS then uses the same prioritization logic as GTO to schedule the occlusion-free warps (❹). Therefore, the younger occlusion-free warps still have a chance to share the cycles with the older warps without memory occlusion. The following sections will detail both static and dynamic OAWS.

4.2 Static OAWS

With the static OAWS policy, we adopt a static miss rate (*SMR*) for each divergent load. The number of MSHR entries required for this load ($Miss_{pred}(pc,w)$) is then predicted as $Div_{pred}(pc,w) \times SMR$, where $Div_{pred}(pc,w)$ is the predicted memory divergence of warp w at load instruction pc. $Div_{pred}(pc,w)$ is equal to the number of active threads in w, similar to DAWS [10], because threads often independently fetch data blocks into L1D when memory divergence occurs. With *SMR* being 0%, OAWS is essentially disabled, and all divergent loads are assumed to complete without consuming MSHR entries. When *SMR* is 100%, OAWS assumes each thread will consume one MSHR entry when divergent loads are issued. We tune static OAWS with *SMR* in the range from 0% to 100% for the optimal performance.

In our experiments, we observe that static OAWS with an *SMR* of 50% achieves the optimal performance on the divergent benchmarks evaluated in this study. Given the SIMD width of 32 in our baseline GPU architecture, 50% means that a warp with 32 active threads is predicted to consume 16 MSHR entries. Since there are only 32 MSHR entries per SM, at most two divergent memory loads can be pipelined into LD/ST units. The qualification logic in Equation 1 then serializes divergent load instructions to access the LD/ST units. Because $Miss_{\text{inflight}}$ is decreased only when a load instruction retires from memory pipeline, this serialization conceptually inserts a minimum delay of $Div_{pred}(pc,w)$ cycles before a new divergent load can be qualified. When the miss rate is high, e.g., when the remaining MSHR entries are less than 16, no warp with 32 threads can be scheduled to issue divergent loads and the delay is further extended. Such a serialization delay reduces the frequency of issuing divergent load instructions and also prevents divergent loads from issuing when available MSHR entries are low, thus static OAWS can alleviate the problem of memory occlusion.

To implement static OAWS, we only need to know whether a load is divergent or not. This information is provided by *DLC*. In general, each *DLC* entry records the history of a divergent load, including *PC* address, the number of instruction occurrences (*#inst*), the total number of memory accesses (*#acc*), and cache associativity contention statistics (*#sets*). When a load instruction's memory accesses are coalesced (❺), its *PC* is first checked against *DLC* to make sure that no duplicate records exist in the table. If a new divergent load is detected, a new entry with the current instruction's

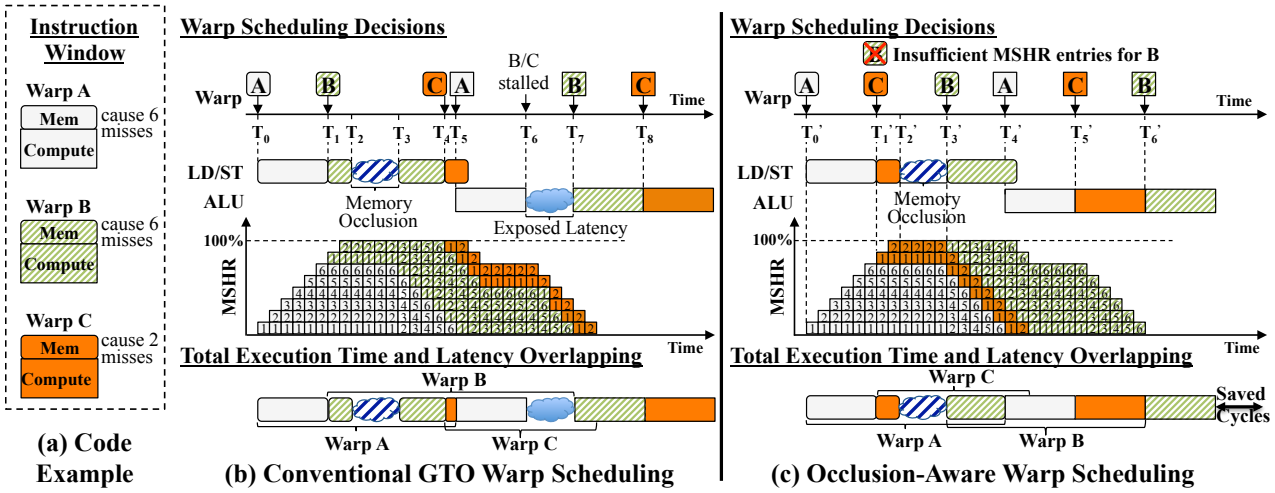

Figure 5: A conceptual example showing the benefit of Occlusion Aware Warp Scheduling

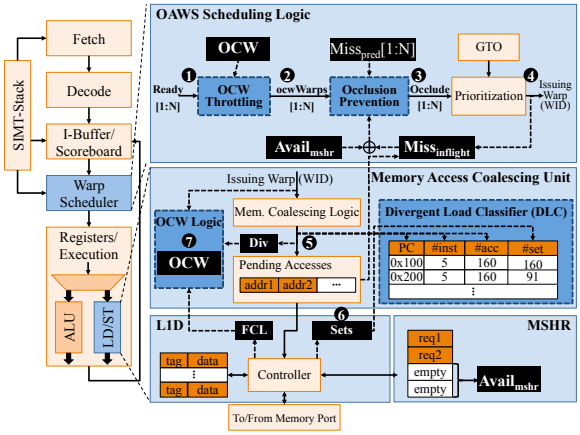

Figure 6: Detailed SM model for our solution. N is the number of warps on each scheduler.

PC, *#inst* (being 1), and *#acc* are inserted into *DLC*; otherwise, the existing entry is updated with this information. *#sets* is updated with the number of cache sets touched by the current load after its requests are processed by L1D(❻). Since OAWS only focuses on divergent loads, coherent loads and their information are not populated into *DLC*. Static OAWS only needs the *PC* field to decide if a load is divergent, while other fields are used by dynamic OAWS.

4.3 Dynamic OAWS

Static OAWS uses a fixed miss rate for all warps, which cannot account for the dynamic nature of divergent accesses across different warps. In addition, memory divergent benchmarks often have high intra-warp locality. Thus, we propose dynamic OAWS to accomplish two objectives: (1) maximizing the number of concurrent warps while preserving L1D cache locality, and (2) maximizing the use of MSHR entries without memory occlusion. This requires a careful selection of appropriate warps that can strike a balance between the maximum resource utilization (cache capacity and MSHR entries) and the best performance (neither memory occlusion nor cache thrashing).

Dynamic OAWS achieves both objectives by enabling a concurrency level at which the aggregate MSHR consumption from all actively scheduled warps is less than MSHR capacity. In order to implement dynamic OAWS, we first propose a light-weight concurrency model to estimate the maximal number of warps that would not occlude the LD/ST units, and then apply this model to predict MSHR consumption on a per-warp basis so that more warps can be opportunistically scheduled.

4.3.1 *Estimation of Optimal Warp Concurrency*

Divergent benchmarks with high intra-warp locality can benefit from reduced cache contention via concurrency throttling. Meanwhile, under the lock-step execution model, warps that incur cache misses, regardless of the number of missed accesses, will inevitably consume MSHR entries and also thrash other warps' cache blocks. We propose to estimate the optimal number of cached warps based on the consumption of cache blocks by load instructions. Note that being fully cached means that a load's data is completely reserved in L1D and will incur no cache eviction in its re-references. Under GTO prioritization, only a few old warps are actively scheduled, as shown in figure 4, thus monitoring the dynamic changes of fully cached loads gives a close approximation of **Optimal Cached Warps (OCW)**.

We also observe that L1D locality in the memory intensive benchmarks can be associated with both coherent and divergent loads. Since coherent loads often exhibit streaming-like accesses and divergent loads tend to dominate the memory footprint, we only use the locality statistics from divergent loads to estimate OCW.

Figure 7 shows the flow of OCW estimation. In the figure, *OCW* denotes the estimated number of fully cached warps, and *CNT* is a counter used to track the changes in the number of fully cached loads. In our proposal, OCW is no less than 2, because each warp scheduler needs at least one active warp (i.e., the oldest warp in GTO) to reduce the idle cycles in warp schedulers; while *Wmax* is the maximum of OCW, which is the number of physical warps on each SM (48 in our baseline). *Cmax* is the maximum of CNT. We empirically use an 8-bit counter for CNT so that it can record the locality changes in 255 consecutive occurrences of divergent loads, i.e., Cmax=255 in Figure 7. OCW estimation is a component inside MACU, denoted as *OCW Logic* (❼) in Figure 6, and is triggered after the accesses of a divergent load are serviced by L1D. CNT is initialized as 128 while OCW is 2 for each SM. These initial values give OCW estimation the flexibility to gradually learn L1D locality changes at runtime.

At runtime, CNT is increased by 1 for each fully cached divergent load (Ⓐ). When CNT is saturated (CNT==Cmax), if OCW has not reached its maximum value (*Wmax*), OCW is increased by

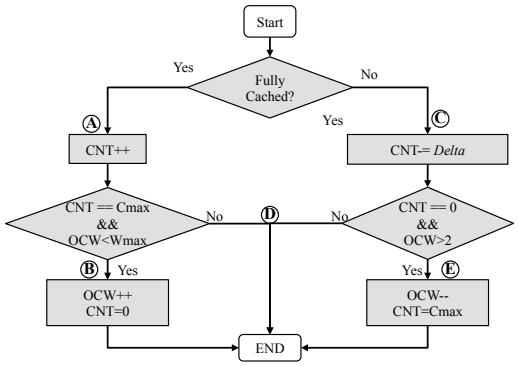

Figure 7: Flow Chart of OCW estimation logic.

1 and accordingly CNT is reset as *0* to track the changes of the fully cached divergent loads under the increased concurrency (Ⓑ). For partially cached loads (Ⓒ), CNT is decreased by *Delta*, where *Delta* is a tuning parameter. When CNT reaches zero, OCW is decreased by 1 so that less warps can be actively scheduled (Ⓔ), and CNT is reset as Cmax. In the corner cases where OCW is equal to 2 or Wmax (Ⓓ), CNT will not be updated if CNT is equal to 0 or Cmax.

To Determine Delta, we have evaluated several heuristics in the tuning process of OCW estimation. Two representative strategies are 1) Delta=1 and 2) Delta=CNT/2. The major difference between the two is that the second strategy can quickly respond to cache contention and bursty MSHR consumption if OCW is overestimated. We observe that cache associativity-sensitive benchmarks, such as these memory-divergent benchmarks from Mars [22] benchmark suite, perform better under the second strategy, but the other memory-divergent benchmarks perform better under the first strategy. For these associativity-sensitive benchmarks, the bursty accesses of each divergent load are concentrated into only a few cache sets [1, 23], severely under-utilizing the L1D capacity. The second strategy then ensures that the estimated OCW can be quickly decreased after a few consecutive occurrences of partially cached divergent loads. We use DLC to track the sensitivity of individual benchmarks to associativity. Ideally, when *#acc* of a divergent instruction is equal to *#sets*, its divergent accesses are uniformly distributed into available cache sets and intra-warp associativity-sensitivity is eliminated. In our implementation, a divergent load is considered as associativity-sensitive if its *#acc* is larger than *#sets*$\times$1.5. Based on this information, we enable strategy 2 for associativity-sensitive loads and strategy 1 for the other loads. Together we can have a robust strategy for Delta.

We use three registers to realize the aforementioned logic of dynamic OAWS. In Figure 6, Register ***Div*** is used to mark whether a load is divergent. Register ***FCL*** is used to track whether a load is fully cached. *FCL* is set when a cache miss happens on it. When all the accesses of the load are serviced, an unset *FCL* indicates a fully cached load. Register ***Sets*** has N-bit, each of which corresponds to one set in the N physical cache sets, and is used to record the number of cache sets touched by the current load. The three registers are reset when a new load instruction is serviced by L1D.

4.3.2 Concurrency-Aware Avoidance of Memory Occlusion

With a scheme from Section 4.3.1 that estimates the optimal cached warps while maintaining L1D cache locality, we can then devise a scheme for our second objective, i.e., maximizing the use of MSHR entries without memory occlusion. We first use the estimated OCW to categorize all warps into two groups: **locality warps** and **thrashing warps**, based on their scheduling priorities. With a descending order for the GTO priorities of warps, a warp is a locality warp if its GTO priority is less than OCW, otherwise a thrashing warp. This classification is based on the observation that older warps are more frequently scheduled in GTO than younger ones so that they are more likely to be fully cached. When a thrashing warp is scheduled, L1D locality is often thrashed.

The estimated OCW can be used to throttle concurrency by only scheduling locality warps, referred to as *OCW Throttling*. However, the training of OCW may take a long time to arrive at the optimal value. This can be a disadvantage for OCW Throttling to quickly respond to program behavior changes during runtime. For example, when L1D locality is well preserved, MSHR consumption from locality warps becomes low; gradually scheduling thrashing warps is then unlikely to immediately incur memory occlusion. Meanwhile, for benchmarks with frequent branch divergences, OCW Throttling may lose the opportunity to schedule thrashing warps with few active threads. Such thrashing warps have small memory footprint, incur little cache thrashing, and consume few MSHR entries. Scheduling them can increase the number of actively scheduled warps. We design dynamic OAWS to opportunistically enable such thrashing warps, when 1) aggregate MSHR consumption of locality warps is low, and 2) some warps have few active threads due to branch divergence.

To prevent memory occlusion while maintaining sufficient concurrency, we integrate the use of OCW in dynamic OAWS for the prediction of cache misses, i.e., the required MSHR entries. For locality warps, we predict their cache misses as 0, i.e., $Miss_{pred}(pc,w) = 0$, because they are fully cached to incur no MSHR consumption. Thus, locality warps will not be excluded from scheduling due to the concern of memory occlusion. Motivated by the little performance fluctuations of per-app SMR with the static OAWS policy, thrashing warps are predicted with a flat miss rate of 50%, i.e., $Miss_{pred}(pc,w) = Div_{pred}(pc,w) \times 50\%$.

A fixed miss rate leads to a uniform prediction for all thrashing warps, unable to capture skewed cache misses from warps with different GTO priorities. As shown in Figure 4, a warp's GTO priority (GTO_{gprio}) is proportional to the number of consumed MSHR entries. In order to enforce the GTO prioritization and provide differentiated predictions for all thrashing warps, we combine both the cache miss rate of a load operation and the GTO_{gprio} of a warp into the predicted misses. Together, we arrive at a formula to predict the required MSHR entries as $Miss_{pred}(pc,w) = Div_{pred}(pc,w) \times 50\% + GTO_{gprio}$.

Combining the estimation of optimal cached warps and the prediction of required MSHR entries, our dynamic OAWS strives to maintain a maximal number of concurrent warps and predict the demands on MSHR entries to prevent memory occlusion. In the integrated scheme for dynamic avoidance of memory occlusion, the OCW Throttling first generates a 2N-bit vector *ocwWarps[1:N]* (❷), in which a value 2 denotes a locality warp, 1 denotes a thrashing warp, and 0 denotes a not-ready warp. Together with the per-warp miss prediction ($Miss_{pred}[1:N]$), the *Occlusion Prevention* component finalizes the qualification logic as shown in Equation 1 to inhibit exhausting MSHR resources and then generates the vector Occlude[1:N] (❸) for the *Prioritization* stage.

4.4 Implementation and Overhead

We summarize other implementation details that have not been covered previously. Note that qualification logic is executed at a per-cycle basis. It is impractical to predict the MSHR consumption for thrashing warps at the same frequency. Thus, we store each instruction's predicted cache misses into the instruction buffer, the same way as the ready bit for baseline qualification logic. By doing so,

Table 1: Baseline GPGPU-Sim Configuration

# of SMs	30 (15 clusters of 2)
SM Configuration	1400MHz, Reg #: 32K, Shared Memory: 48KB, SIMD Width: 16, warp: 32 threads, max threads per SM: 1536
Caches / SM	Data: 32KB/128B-line/8-way, Constant: 8KB/64B-line/24-way, Texture: 12KB/128B-line/2-way
Branch Handling	PDOM based method [24]
Warp Scheduling	GTO
Interconnect	Butterfly, 1400MHz, 32B channel width
L2 Unified Cache	768KB, 128B line, 16-way
Min. L2 Latency	120 cycles (compute core clock)
Cache Indexing	Pseudo-Random Hashing Function [25]
Global Memory	6 partitions, min latency: 100 cycles
Memory Controller	Out-of-Order (FR-FCFS), max request queue length: 32
GDDR5 Timing	nbk=16, $t_{CL}=12$, $t_{RP}=12$, $t_{RC}=40$, $t_{RAS}=28$, $t_{RCD}=12$, $t_{RRD}=6$, $t_{CDLR}=5$, $t_{WR}=12$

Table 2: Memory-Intensive CUDA Benchmarks

#	Name	#	Name	#	Name	#	Name
Memory Divergent Benchmarks							
1	ATAX [28]	4	GES [28]	7	PF [27]	10	KMN [27]
2	BICG [28]	5	SYRK [28]	8	BFS [27]	11	SPMV [29]
3	MVT [28]	6	SYR2K [28]	9	SC [27]	12	IIX [22]
Memory Coherent Benchmarks							
13	3DC [28]	15	SRAD1 [27]	17	BP [27]	19	3MM [28]
14	2MM [28]	16	SRAD2 [27]	18	FD [28]	20	LBM [30]

Table 3: Config. for SWL-Best and CCWS

SWL-Best				CCWS	
Bench.	#Warps	Bench.	#Warps	Name	Value
ATAX	2	SYR2K	2	$K_{THROTTLE}$	8
BICG	2	KMN	4	Victim Tag Array	8-way
MVT	2	BFS	3		16 entries/warp
GES	1	SPMV	2		(768 total entries)
SYRK	2	IIX	4	Warp Base Score	100

the process of miss prediction can be executed off the critical path of warp scheduling.

Divergent Load Classifier. The benchmarks we evaluate in this work typically have only one or two divergent loads in each kernel. But IIX from Mars [22] has 26 divergent loads in one of its kernels. Thus we have 32 entries for DLC in both static and dynamic OAWS implementations. Because the fields of DLC are periodically updated, the entries that reach 0 (#inst) could be evicted. DLC could also be managed under the LRU policy for more complicated workloads. In addition, DLC is cleared at each kernel invocation.

Overhead Analysis. To implement the qualification logic in Equation 1, OAWS stores $Avail_{mshr}$ and $Miss_{inflight}$ in two registers. Our baseline GPU has 32 MSHR entries, and both registers need only 5 bits. OCW estimation needs three registers: 1-bit *Div*, 8-bit *FCL*, and 32-bit *Sets*. Our baseline GPU has a SIMD-width of 32. We use 5 bits to store the predicted cache misses for each instruction. Considering that there are 48 warps per SM and each warp can have two instructions, storing the predicted per-instruction cache misses has a total overhead of 480 bits (60 bytes). Each DLC entry needs 9 bytes, i.e., 40 bits for *PC*, 7 bits for *#inst*, 10 bits for *#acc*, and 10 bits for *#set*. Thus, DLC table needs 288 bytes. In total, implementing OAWS needs 348-byte on-chip storage and five registers in each SM.

5 Experimental Evaluation

We use GPGPU-Sim [26] (version 3.2.1), a cycle-accurate simulator, to evaluate our OAWS mechanisms. The baseline GPU architectural parameters are summarized in Table 1. Highly memory-divergent benchmarks from Rodinia [27], PolyBench [28], SHOC [29], Parboil [30], and Mars [22] are used for performance evaluation. These benchmarks are listed in Table 2. We also evaluate the performance of OAWS on memory-coherent benchmarks from PolyBench/GPU [28]. All of the benchmarks are simulated to completion and execute between 70 million and 1.5 billion instructions. The following warp scheduling algorithms are evaluated:

GTO is the baseline warp scheduler. All performance metrics are normalized to GTO.

SWL [9] statically limits the number of warps that can be actively scheduled and needs to be tuned on a per-benchmark basis. Table 3 presents the warp-limiting value with the best performance for some memory divergent benchmarks (*SWL-Best*).

CCWS [9] relies on a dedicated victim cache and a 6-bit Warp ID field in the tag of cache block to detect intra-warp locality and other storage to track per-warp locality changes. The warp that has the largest locality loss is exclusively prioritized. Configuration parameters for CCWS are summarized in Table 3.

MASCAR [13] exclusively prioritizes memory instructions from one "owner" warp when the memory subsystem is saturated; otherwise, memory instructions of all warps are prioritized over any computation instruction. MASCAR uses a re-execution queue to replay L1D accesses that are stalled due to MSHR unavailability or network congestion. Saturation here means that MSHR has only 1 entry or the queue inside memory port has only 1 slot. The re-execution queue has 32 entries.

Static OAWS (*OAWS-Static*) is described in Section 4.2. The default value for *SMR* is 50%. We will present the sensitivity analysis of *SMR* in Section 5.4.

Dynamic OAWS (*OAWS-Dyn*) is described in Section 4.3. *OAWS-Dyn* consists of two components, light-weight estimation of OCW (Optimal Concurrent Warps) and dynamic MSHR prediction.

5.1 Instructions Per Cycle (IPC)

Figure 8 shows IPC comparisons for our scheduling algorithms and three state-of-the-art algorithms. We have the following key observations. First, *OAWS-Static* consistently improves performance for memory divergent benchmarks, and on average achieves 36.7% IPC gains compared to baseline GTO scheduling and outperforms *MASCAR* by 34.4%. The performance improvement of *OAWS-Static* comes with the lowest hardware overhead, which strongly suggests the need of occlusion prevention. Second, by focusing on locality changes at the granularity of individual divergent loads, *OAWS-Dyn* dynamically determines the OCW value, therefore it can outperform *CCWS* and *SWL-Best* by 57.2% and 10.9%, respectively. Lastly, *OAWS-Dyn* effectively increases the performance by opportunistically scheduling thrashing warps. Overall, *OAWS-Dyn* achieves 73.1% IPC improvement and outperforms *MASCAR*, *CCWS*, *SWL-Best* by 70.1%, 57.8%, and 11.4%, respectively.

MASCAR and *CCWS* only improve performance by 1.7% and 9.7% respectively compared to the baseline. This low IPC gains can be explained from the following two aspects. First, we use an allocate-on-fill rather than an allocate-on-miss policy to manage L1D blocks on cache read misses. Given a 32-entry MSHR and 32KB L1D (256 blocks), the allocate-on-miss policy frequently reserves 32 cache blocks for outstanding memory requests in the memory divergent benchmarks, which wastes 12.5% of the L1D

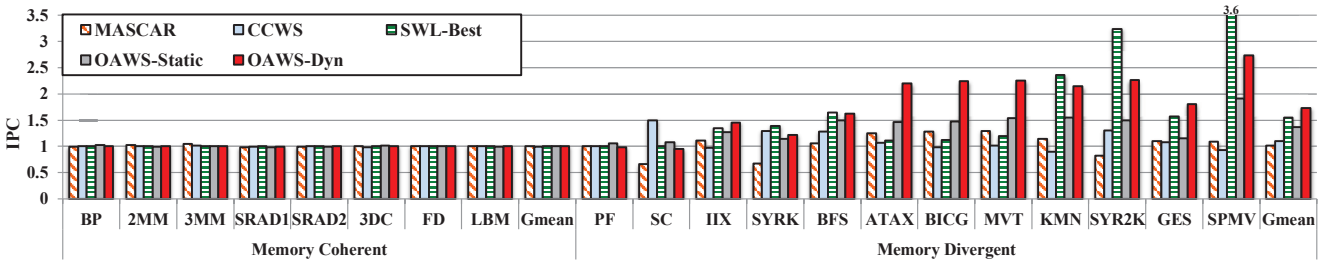

Figure 8: IPCs of various warp scheduling algorithms for memory coherent and memory divergent benchmarks. IPCs are normalized to the GTO scheduling.

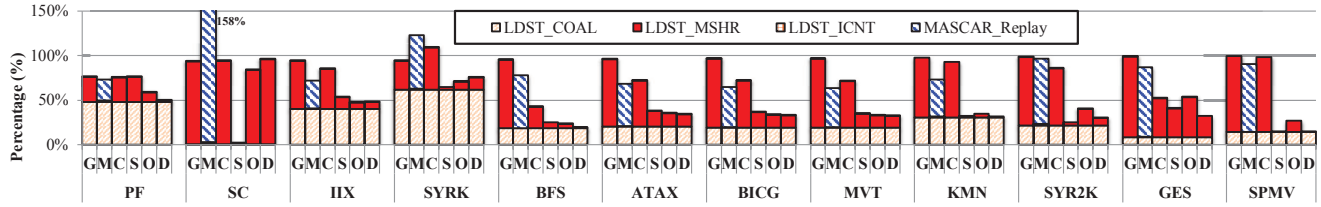

Figure 9: Breakdown of LD/ST stall cycles when the memory divergent benchmarks are scheduled by GTO (G), MASCAR (M), CCWS (C), SWL-Best (S), OAWS-Static (O), and OAWS-Dyn (D). *MASCAR_Replay* only exists in MASCAR and refers to the cycles when the memory access from the re-execution queue cannot be sent out.

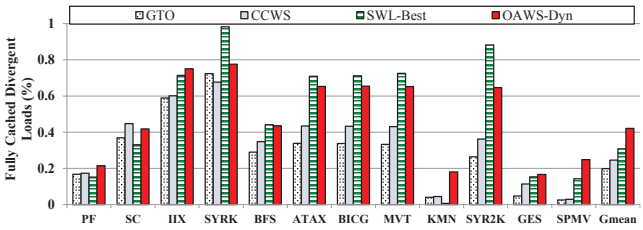

Figure 10: Percentage of fully cached divergent loads in the memory divergent benchmarks.

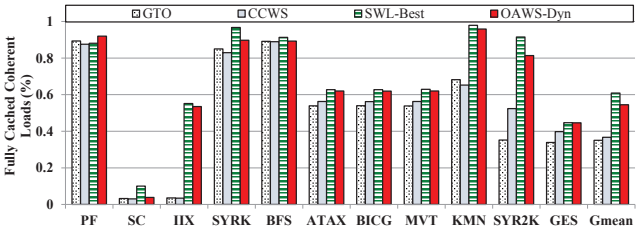

Figure 11: Percentage of fully cached coherent loads in the memory divergent benchmarks. SPMV has no coherent loads and is excluded.

capacity. Second, reserving cache blocks can increase associativity conflicts in L1D. The memory divergent benchmarks from PolyBench/GPU [28] are highly sensitive to cache associativity [1]. Though we have applied the pseudo-random cache indexing function that is used in Fermi architectures [25], associativity conflicts are not well mitigated. For example, each divergent load in ATAX generates 32 requests that are constantly mapped into 8 out of the 32 sets in L1D. Consequently, allocate-on-miss aggravates associativity conflicts. We have evaluated both policies for all benchmarks, as shown in Figure 14, and observe that *MASCAR* performs similarly between the two and *CCWS* favors allocate-on-miss. However, GTO scheduling benefits greatly from allocate-on-fill because of the increased effective L1D capacity and decreased associativity conflicts.

Although *SWL-Best* achieves 55.5% IPC improvement for all memory divergent benchmarks, it is a profile-based static approach that requires tuning experiments for each GPU kernel with every data input. Thus *SWL-Best* is impractical to be implemented in GPU, but it could be a reference to gauge the effectiveness of GPU scheduling algorithms. In our experiments, *OAWS-Dyn* outperforms *SWL-Best* on most of memory divergent benchmarks, except *SYR2K* and *SPMV* benchmarks. This is because of the unique behaviors of *SYR2K* and *SPMV*. *SYR2K* has no branch divergences and very high inter-warp locality, and *SPMV* with little coherent load operations. OAWS first emphasizes a policy to predict and control the use of MSHR entries based on load divergence, then the concurrency level. This causes OAWS to have different decisions than SWL-best, on the set of active warps for SYR2K and SPMV. The performance of *SWL-Best* could have been better if per-kernel tuning had been performed. For example, ATAX, BICG, and MVT have both coherent and divergent kernels. *SWL-Best*'s performance gains in divergent kernels are balanced out by the existence of coherent kernels where *SWL-Best* should not have been enabled. Meanwhile, *IIX* is a highly complicated benchmark with rich branch- and memory-divergence. IIX has 149 kernel invocations with the input size used in this evaluation. Given such complexity, *SWL-Best* has been included for pure performance comparison purposes. For the coherent benchmarks, none of the warp scheduling algorithms achieve significant performance gains. This indicates that our OAWS has no detrimental effects to memory-coherent benchmarks. In the following sections, we will dissect the performance of both *OAWS-Static* and *OAWS-Dyn* using memory divergent benchmarks only.

5.2 LD/ST Unit Stalls

Since the L1D cache misses come from three portions as shown in Figure 2, simply quantifying the accuracy of our MSHR estimation cannot directly determine cache misses. In Figure 9, we break down LD/ST stall cycles when memory divergent benchmarks are scheduled by various scheduling algorithms. All numbers are normalized to LD/ST active cycles in GTO scheduling. Since all the evaluated benchmarks are read-intensive, memory occlusion caused by network congestion (*LDST_ICNT*) is negligible in every scheduling algorithm. Meanwhile, coalescing stalls (*LDST_COAL*) depend on memory divergence characteristics in the benchmarks. Warp scheduling can only impact stall cycles caused by *LDST_MSHR* and, in MASCAR by by re-execution queue de-

lays (*MASCAR_Replay*). At the same time, *LDST_MSHR* stalls are largely reduced by our scheme since the stalls in LD/ST units are overlapped with computations in the execution units. As we can see from Figure 9, *SWL-Best*, *OAWS-Static*, and *OAWS-Dyn* have similar capability in reducing LD/ST stalls, which corresponds to their superior performance shown in Figure 8. Divergent loads in SC are references to arrays of structs and outside of a loop that generates high locality. Without careful concurrency throttling, divergent loads quickly thrash the locality of coherent loads. *OAWS-Dyn* takes time to learn optimal concurrency, while *OAWS-Static* has no direct control over concurrency, thus they perform equally poor in reducing LD/ST stalls for SC. When saturation in the memory subsystem is detected, MASCAR prevents memory accesses of "non-owner" warps from being sent out, i.e., replaying them until saturation is resolved. Such a strict requirement directly reduces LD/ST throughput, leading to the poor performance of MASCAR. *CCWS* prioritizes warps with the highest locality lost, which means that the prioritized warp often has little data reserved in L1D to start with and needs to immediately fetch data from L2. Switching prioritized warps incurs frequent MSHR consumptions, making CCWS less capable of LD/ST stall prevention.

5.3 Fully Cached Load Instructions

Within the SIMD execution, partially cached instructions still suffer from the memory occlusion. More fully cached loads create an opportunity for warp schedulers to better utilize any localities in L1D. Thus we use the number of fully cached loads to quantify effective concurrency in the evaluated concurrency throttling mechanisms, i.e., *CCWS*, *SWL-Best*, and *OAWS-Dyn*. Figure 10 presents the percentages of fully cached divergent loads. The results of GTO are also included as the baseline. *CCWS* has increased fully cached loads for all benchmarks except SYRK. Because SYRK has high inter-warp locality, while *CCWS* is specifically designed for intra-warp locality protection. The exclusively prioritized warp can accelerate progress and evict its own data blocks that could have been utilized by other warps. *SWL-Best* achieves the best results for benchmarks with no branch divergence, such as SYRK, ATAX, BICG, MVT, and SYR2K. Due to high inter-warp locality, *SWL-Best* fully caches 98% and 88% of divergent loads in SYRK and SYR2K, respectively. For memory- and branch-divergent benchmarks, such as IIX and BFS, *OAWS-Dyn* preserves more fully cached divergent loads than *SWL-Best*. On average, GTO, *CCWS*, *SWL-Best*, and *OAWS-Dyn* keep 20%, 25%, 31%, and 45% of the total divergent loads in L1D cache, respectively.

Figure 11 presents the percentages of fully cached coherent loads. Some coherent loads in SC exhibit streaming accesses, achieving very low percentages of fully cached coherent loads in all cases. IIX has observable program behavior changes, i.e., coherent memory operations and large computation strictly follow divergent memory operations. Thus *OAWS-Dyn* preserves more coherent loads in L1D, which explains its performance advantage in IIX as shown in Figure 8. The other benchmarks are relatively simple. Their trend of fully cached loads is similar to that in Figure 10. Results in Figures 10 and 11 are highly correlated to the IPC discrepancy in Figure 8, except that *SWL-Best* does not have the best performance in ATAX, BICG, and MVT. As mentioned earlier, this is due to the fact that the three benchmarks have both memory divergent and memory coherent kernels.

5.4 Sensitivity of Static OAWS

Figure 12 presents the IPC of static OAWS when *SMR* is swept from 0% to 100%, with an increment of 5%. When *SMR* is 0%, static OAWS is equal to GTO. Static OAWS achieves peak performance improvement (geometric mean: 35.3%), when *SMR* is

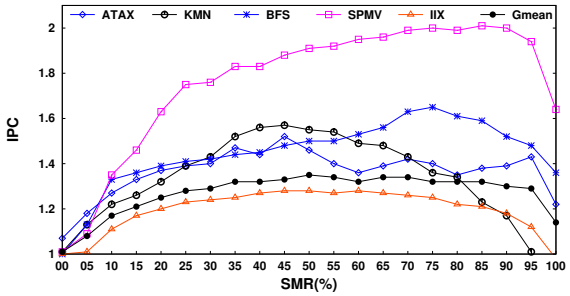

Figure 12: IPC of static OAWS with various *SMR* under five representative benchmarks.

50%. In addition, when *SMR* ranges from 35% to 90%, the OAWS achieves stable IPC improvement with the standard deviation of 0.014, indicating its insensitivity to *SMR*. This is because with *SMR* larger than 50%, the qualification logic of static OAWS prevents two fully divergent loads from being issued, leading to such similar IPC improvements.

5.5 Sensitivity to MSHR Sizes

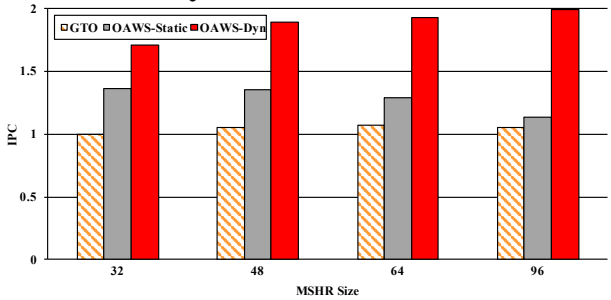

Figure 13: IPC of dynamic OAWS with different MSHR sizes on divergent benchmarks.

Figure 13 shows the IPC improvement (GMean) from three different policies with different MSHR sizes. All results are normalized to GTO with 32 entries and the allocate-on-fill policy. Several interesting trends can be observed. GTO cannot benefit from more MSHR entries, which indicates that the memory occlusion issue cannot be simply addressed by the provision of more resources. Our investigation suggests that on-chip interconnect plays a critical role in this. Adding more MSHRs will increase the volume of bursty in-flight memory requests to the off-chip memory chips, which will congest the on-chip interconnect and prolong memory accesses. In addition, the static scheme *OAWS-Static* with an SMR of 50% cannot benefit from more MSHR entries either. Counter-intuitively, it experiences performance degradation with an increasing number of MSHR entries. This is actually caused by the mismatch between the available MSHR entries and the static prediction. An increasing number of MSHR entries cause the static OAWS policy to schedule more thrashing warps, contending for the limited L1D. In contrast, *OAWS-Dyn* delivers an increasing amount of performance improvement with more MSHR entries. The key for the performance improvement of *OAWS-Dyn* is leveraging additional entries for the load instructions with less divergence and better locality. Therefore, *OAWS-Dyn* alleviates the situation by controlling divergent loads and prioritizing coherent loads for the use of MSHR entries, leading to the better performance and effective concurrency. Thus *OAWS-Dyn* can improve performance by 73.1%, 89.1%, 92.6%, and 99.4% when there are 32, 48, 64, and 96 MSHR entries, respectively. It successfully corrects the problem

of memory occlusion by dynamically predicting an optimal number of cached warps and effectively leveraging more MSHR entries without thrashing L1D.

5.6 Sensitivity to L1D Allocate Policy

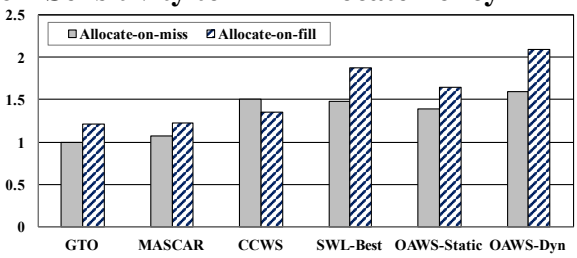

Figure 14: IPC of various scheduling techniques to L1D allocate policies on divergent benchmarks.

Figure 14 provides a comparison of two L1D allocate policies: allocate-on-miss and allocate-on-fill. All results are normalized to allocate-on-miss, the default policy in GPGPU-Sim [26]. GTO achieves 20.9% better performance when the L1D is managed by allocate-on-fill. Among the 7 scheduling techniques, only CCWS favors allocate-on-miss. Although this policy increases cache contention by reserving cache lines for outstanding memory requests, CCWS provides longer prioritization for the warp that has the largest locality loss, leading to 15.5% IPC improvement.

6 Related Work

Warp Scheduling plays a critical role in sustaining GPU performance and various scheduling algorithms have been proposed based on different heuristics. Among concurrency throttling techniques, SWL [9], CCWS [9], and MASCAR [13] were discussed earlier in Section 5 and compared with our proposed scheduling mechanisms. On top of CCWS, DAWS [10] actively schedules warps whose aggregate memory footprint does not exceed L1D capacity. Khairy *et al.* [8] proposed DWT-CS, which use cores sampling to throttle concurrency. When L1D MPKI is above a given threshold, DWT-CS samples all SMs with different number of active warps and applies the best-performing active warp count on all SMs. Different from these concurrency throttling mechanisms, OAWS uses the number of fully cached divergent load instructions to dynamically adjust concurrency at a finer-granularity. CTA scheduling techniques [31, 32, 20] are coarser than warp scheduling at concurrency throttling, thus OAWS is better at locality preservation and LD/ST stall avoidance.

Some other warp scheduling algorithms are designed to improve GPU resource utilization. Fung *et al.* [24, 33] investigated the impact of warp scheduling on techniques aiming at branch divergence reduction, i.e., dynamic warp formation and threadblock compaction. Narasiman *et al.* [19] proposed a two-level round robin scheduler to prevent memory instructions from being issued consecutively. By doing so, memory latency can be better overlapped by computations. Gebhart *et al.* [34] introduced another two-level warp scheduler to manage a hierarchical register file design. None of these warp scheduling directly focuses on the problem of LD/ST stalls. On top of the two-level warp scheduling, Yu *et al.* [35] proposed a Stall-Aware Warp Scheduling (SAWS) to adjust the fetch group size when pipeline stalls are detected. SAWS mainly focuses on pipeline stalls, while OAWS is capable of avoiding LD/ST stalls and preserving L1D locality.

Kayiran et al. [31] proposed a dynamic Cooperative Thread Array (CTA) scheduling mechanism to enable the optimal number of CTAs according to application characteristics. It reduces concurrent CTAs for memory-intensive applications to reduce LD/ST stalls. Lee et al. [32] proposed two alternative CTA scheduling schemes. Lazy CTA scheduling (LCS) utilizes a 3-phase mechanism to determine the optimal number of CTAs per core, while Block CTA scheduling (BCS) launches consecutive CTAs onto the same cores to exploit inter-CTA data locality.

GPU Cache Management has been studied to preserve L1D locality, which can implicitly reduce LD/ST stalls. And L1D bypassing is often adopted to alleviate cache contention [1, 2, 3, 4, 5, 6, 7, 8, 36, 37]. Jia *et al.* [1] designed a memory request buffer to reorder and prioritize L1D accesses, and proposed to bypass L1D accesses that are stalled by cache associativity conflicts. Chen *et al.* [2] used extensions in L2 cache tag to track locality loss in L1D, and bypass is temporarily triggered if a L2 cache block is requested twice by the same SM. Chen *et al.* [3] further proposed coordinated bypassing and warp throttling (CBWT) to orchestrate L1D bypassing and warp scheduling. Based on protection distance prediction [38], CBWT triggers bypassing when all L1D blocks are under protection and throttles concurrency to prevent NOC from being congested by aggressive bypassing. Wang *et al.* [4] proposed a DaCache design to orchestrate GPU cache management and warp scheduling. At runtime, DaCache bypasses divergent loads from warps with low scheduling priorities, and coherent loads with no locality. Dong Li proposed an AgeLRU algorithm [6] to prevent young warps from evicting blocks of old warps. AgeLRU enables bypassing when the replacement score of the replacement candidate is above a given threshold. Based on DAWS, Zheng *et al.* [7] proposed Adaptive Cache and Concurrency (CCA) to bypass streaming memory accesses and accesses from inactive warps. Similarly, Khairy *et al.* [8] also proposed a technique to dynamically detect and bypass streaming memory accesses. Though these cache management schemes can ameliorate the problem of LD/ST stalls via preserved L1D locality, they are all reactive mechanisms. OAWS works from the source to prevent LD/ST stalls from occurring. Besides, cache management schemes are orthogonal to OAWS and can be combined with OAWS to further improve GPU performance.

7 Conclusion

We identified the occlusion of LD/ST units and the depletion of MSHR entries caused by divergent memory accesses in the GPU execution pipeline, which was referred to as memory occlusion. We then characterized and analyzed its impact. To address these issues, we proposed memory occlusion aware warp scheduling that can predict the demand of MSHR entries from GPU instructions and integrate this information into the qualification stage of warp schedulers to prevent memory occlusion. Both static and dynamic prediction methods were designed and implemented to maximize the number of concurrent warps without memory occlusion. We further evaluated OAWS with static and dynamic methods on a wide variety of memory benchmarks. Compared to the default GTO, static and dynamic OAWS policies achieved 36.7% and 73.1% performance gains, respectively. Compared to state-of-the-art warp schedulers, i.e., MASCAR [13], CCWS [9], and SWL-Best [9], dynamic OAWS outperformed them by 70.1%, 57.8%, and 11.4%, respectively. The benefits come from the reduced MSHR stalls as shown in Figure 9 and the increased number of fully cached loads as demonstrated in Figure 10.

Acknowledgment

We are very thankful for the insightful comments from the anonymous reviewers. This work is supported in part by National Science Foundation awards 1059376, 1340947, 1561041, and 1564647.

8 References

[1] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *HPCA*, 2014.

[2] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W. mei W. Hwu, "Adaptive Cache Bypass and Insertion for Many-core Accelerators," in *MES*, 2014.

[3] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. mei W. Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *MICRO*, 2014.

[4] B. Wang, W. Yu, X.-H. Sun, and X. Wang, "DaCache: Memory Divergence-Aware GPU Cache Management," in *ICS*, 2015.

[5] C. Li, S. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-Driven Dynamic GPU Cache Bypassing," in *ICS*, 2015.

[6] D. Li, *Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processor*. PhD thesis, University of Texas at Austin, May 2014.

[7] Z. Zheng, Z. Wang, and M. Lipasti, "Adaptive Cache and Concurrency Allocation on GPGPUs," *Computer Architecture Letters*, 2014.

[8] M. Khairy, M. Zahran, and A. G. Wassal, "Efficient Utilization of GPGPU Cache Hierarchy," in *GPGPU*, 2015.

[9] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[10] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware Warp Scheduling," in *MICRO*, 2013.

[11] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization," in *ISCA*, 1981.

[12] A. E. Turner, *On replay and hazards in graphics processing units*. PhD thesis, University of British Columbia, Oct 2012.

[13] A. Sethia, D. A. Jamshidi, and S. A. Mahlke, "Mascar: Speeding up GPU Warps by Reducing Memory Pitstops," in *HPCA*, 2015.

[14] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, pp. 39–55, Mar. 2008.

[16] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[17] B. Coon, P. Mills, S. Oberman, and M. Siu, "Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequenti al register size indicators," Oct. 7 2008. US Patent 7,434,032.

[18] P. Mills, J. Lindholm, B. Coon, G. Tarolli, and J. Burgess, "Scheduler in multi-threaded processor prioritizing instructions passing qualification rule," May 24 2011. US Patent 7,949,855.

[19] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*, 2011.

[20] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[21] N. Brunie, S. Collange, and G. F. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," in *ISCA*, 2012.

[22] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce Framework on Graphics Processors," in *PACT*, 2008.

[23] B. Wang, Z. Liu, X. Wang, and W. Yu, "Eliminating intra-warp conflict misses in GPU," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pp. 689–694, 2015.

[24] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO*, 2007.

[25] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A Detailed GPU Cache Model Based on Reuse Distance Theory," in *HPCA*, 2014.

[26] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[27] 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 *IISWC*, 2009.

[28] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes.," in *Innovative Parallel Computing*, 2012.

[29] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *GPGPU*, 2010.

[30] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Changx, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign*, 2012.

[31] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.

[32] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y.-G. Cho, and S. Ryu, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *HPCA*, 2014.

[33] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control flow," in *HPCA*, 2011.

[34] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *ISCA*, 2011.

[35] Y. Yu, W. Xiao, X. He, H. Guo, Y. Wang, and X. Chen, "A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-level Parallelism in GPGPUs," in *ICS*, 2015.

[36] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-fetched Caches in GPUs," in *ICS*, 2012.

[37] X. Xie, Y. Liang, G. Sun, and D. Chen, "An Efficient Compiler Framework for Cache Bypassing on GPUs," in *ICCAD*, 2013.

[38] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *MICRO*, 2012.