Mascar: Speeding up GPU Warps by Reducing Memory Pitstops

Ankit Sethia, D. Anoushe Jamshidi and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI
{asethia, ajamshid, mahlke}@umich.edu

***Abstract*—With the prevalence of GPUs as throughput engines for data parallel workloads, the landscape of GPU computing is changing significantly. Non-graphics workloads with high memory intensity and irregular access patterns are frequently targeted for acceleration on GPUs. While GPUs provide large numbers of compute resources, the resources needed for memory intensive workloads are more scarce. Therefore, managing access to these limited memory resources is a challenge for GPUs. We propose a novel Memory Aware Scheduling and Cache Access Re-execution (Mascar) system on GPUs tailored for better performance for memory intensive workloads. This scheme detects memory saturation and prioritizes memory requests among warps to enable better overlapping of compute and memory accesses. Furthermore, it enables limited re-execution of memory instructions to eliminate structural hazards in the memory subsystem and take advantage of cache locality in cases where requests cannot be sent to the memory due to memory saturation. Our results show that Mascar provides a 34% speedup over the baseline round-robin scheduler and 10% speedup over the state of the art warp schedulers for memory intensive workloads. Mascar also achieves an average of 12% savings in energy for such workloads.**

I. Introduction

With the popularization of high level programming models such as CUDA [24] and OpenCL [17], there has been a proliferation of general purpose computing on GPUs to exploit their computational power. While OpenCL and CUDA enable easy portability to GPUs, significant programmer effort is necessary to optimize the programs and achieve near peak performance of the GPU [29], [31]. This effort is increased when data parallel workloads that are memory intensive are targeted for acceleration on GPUs due to their parallel processing paradigm and are still far from achieving the potential performance offered by a GPU.

Modern GPUs utilize a high degree of multi threading in order to overlap memory accesses with computation. Using the Single Instruction Multiple Thread (SIMT) model, GPUs group many threads that perform the same operations on different data into warps, and a warp scheduler swaps warps that are waiting for memory accesses for those that are ready for computation. One of the greatest challenges that prevents achieving peak performance on these architectures is the lack of sufficient computations. When a memory intensive workload needs to gather more data from DRAM for its computations, the resources for managing outstanding memory requests (at the L1, L2, and memory subsystem) become saturated. Due to this saturation, a new memory request can only be sent when older memory requests complete. At this point, subsequent accesses to the memory can no longer be pipelined, resulting in serialized accesses. In such a scenario, the computation portion of any of the parallel warps cannot begin, as all the memory requests needed to initiate the computation have been serialized. Furthermore, the amount of computation to hide the latency of the unpipelined memory requests is much larger. For such memory constrained workloads, a GPU's compute units are forced to wait for the serialized memory requests to be filled, and the workloads cannot achieve high throughput.

Another of memory intensive applications saturating memory subsystem resources is the reduction in data reuse opportunities for the L1 cache. Due to the saturated memory pipeline, including the Load Store Unit (LSU), warps whose data are present in L1 cannot issue memory instructions. This delay in utilizing the already present data may result in the eviction of the reusable data by other warp's data being brought in to the cache. This leads to increased cache thrashing, forcing more DRAM requests to be issued, thus worsening the serialization of memory accesses. While ideal GPU workloads tend to have very regular, streaming memory access patterns, recent research has examined GPU applications that benefit from cache locality [27], [19] and have more non-streaming accesses. If this data locality is not exploited, cache thrashing will occur, causing performance degradation.

In this work, we establish that the warp scheduler present in a GPU's Streaming Multiprocessor (SM) plays a pivotal role in achieving high performance for memory intensive workloads, specifically by prioritizing memory requests from one warp over those of others. While recent work by Jog et al. [15] has shown that scheduling to improve cache and memory locality leads to better performance, we stipulate that the role of scheduling is not limited to workloads which have such locality. We show that scheduling is also critical in improving the performance of many memory intensive workloads that do not exhibit data locality.

We propose Memory Aware Scheduling and Cache Access Re-execution (Mascar) to better overlap computation and memory accesses for memory intensive workloads. The intuition behind Mascar is that when the memory subsystem is saturated, all the memory requests of one warp should be prioritized rather than sending a fraction of the required requests from all warps. As the memory subsystem saturates, memory requests are no longer pipelined and sending more requests from different warps will delay any single warp from beginning computation. Hence, prioritizing all requests from one warp allows this warp's data to be available for computation sooner, and this computation can now overlap with the memory accesses of another warp.

While Mascar's new scheduling scheme enables better overlapping of memory accesses with computation, memory subsystem saturation can also prevent reuse of data in the L1 cache. To ameliorate this, we propose to move requests stalled in LSU due to memory back pressure to a re-execution queue where they will be considered for issuing to the cache at a later time. With such a mechanism, the LSU is free to process another warp whose requested addresses may hit in the cache. Re-execution can both improve cache hit rates by exploiting such hits under misses, allowing the warp to now execute computation, as well as reduce back pressure by preventing this warp from accessing DRAM twice for the same data.

This paper makes the following contributions:

- We analyze the interplay between workload requirements, performance, and scheduling policies. Our results show that the choice of scheduling policy is critical for memory intensive workloads, but has lesser impact on the performance of compute intensive workloads.

- We propose a novel scheduling scheme that allows better overlapping of computation and memory accesses in memory intensive workloads. This scheme limits warps that can simultaneously access memory with low hardware overhead.

- We also propose a memory instruction re-execution scheme. It is coupled with the LSU to allow other warps to take advantage of any locality in the data cache, when the LSU is stalled due to memory saturation.

- An evaluation of Mascar on a model of the NVIDIA Fermi architecture achieves 34% performance improvement over baseline GPU for workloads sensitive to scheduling while reducing average energy consumption by 12%.

II. Background and Motivation

A. Background

Modern GPUs are comprised of numerous streaming multiprocessors (SMs), each of which execute in parallel. Figure 1 illustrates a high-level view of the hardware considered in this work. The details inside the SM, which include the execution units, register file, etc., have been abstracted as we focus on the memory subsystem of the GPU. Each SM has a private L1 cache, which uses miss status holding registers (MSHRs) to permit a limited number of outstanding requests, as shown in the figure. All SMs can communicate with a unified L2 cache using the interconnect. If the request misses in the L2 cache, it will go off-chip using a DRAM channel as shown in Figure 1. Like the L1 cache, each L2 partition can also handle a fixed number of outstanding memory requests through its MSHRs.

Memory intensive workloads will generate a very large number of memory requests. This typically results in the MSHRs in the L2 filling up much faster than when compute intensive workloads are run. The L2 is then forced to stop accepting requests and sends negative acknowledgements to new requests coming through the interconnect. As the L2 rejects any new requests, the buffers in the interconnect and between the L2 and L1 caches will begin to fill up. When these buffers are full, the interconnect will cease to accept new requests from a SM's L1. Any new request coming from the SM is allocated a new MSHR at L1, but no new request can be

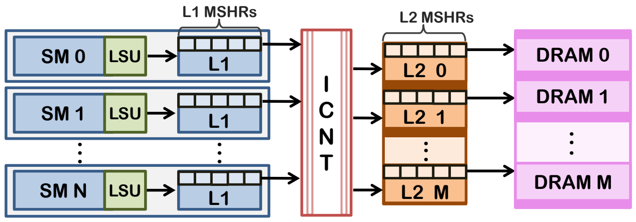

Fig. 1: A diagram of the GPU memory hierarchy, with N SMs and M memory partitions. Each SM has a Load-Store Unit (LSU) that issues memory requests. The L1 & L2 caches support a limited number of outstanding misses using MSHRs.

sent to the L2 due to the earlier congestion in the interconnect. The cascading effect of the rejection of memory requests will reach the SM once all of its L1's MSHRs have been reserved. At that point an SM can no longer issue new memory instructions from any ready warp. This results in serialization of all new memory requests as they can only be sent if an earlier request is completed in the memory subsystem, which will free up one of the MSHRs and begin to relieve the back pressure throughout the rest of the subsystem's resources.

If the LSU begins to stall due to memory back pressure, no warp can access the L1 cache until an MSHR is available. Other warps that need the SM's compute units can still continue execution. If all the memory requests of a warp return before other warps finish executing computational code, then this now ready to execute warp can help to hide the latency of other warps whose memory requests have not yet returned. However, due to the serialization of memory accesses, the amount of computation required to mask the remaining memory latency is significantly higher than the amount required when the memory requests are pipelined.

B. Motivation

To identify data parallel application kernels that suffer due to the memory subsystem back pressure issues described in Section II-A, we classify kernels from the Rodinia [5] and Parboil [33] benchmark suites as compute or memory intensive. For each of these kernels, Figure 2 shows the fraction of the theoretical peak IPC achieved (left bar) and the fraction of cycles for which the SM's LSU is forced to stall (right bar) due to memory subsystem saturation[1].

Of the 30 kernels in the two benchmark suites, 15 are in the compute intensive category whereas 15 are considered memory intensive. The kernels are arranged in decreasing order of their fraction of peak IPC achieved within their category. While 13 out of 15 kernels in the compute intensive category achieve more than 50% of the peak IPC for compute intensive kernels, only one out of the 15 kernels in the memory intensive category achieves 50% of the peak IPC. In fact, eight kernels achieve less than 20% of the peak performance in this category, whereas no kernel in the compute intensive category suffers from such low performance.

Figure 2 illustrates the strong correlation between the performance achieved and the memory demands of the GPU

[1]The methodology for this work is detailed in Section IV-A.

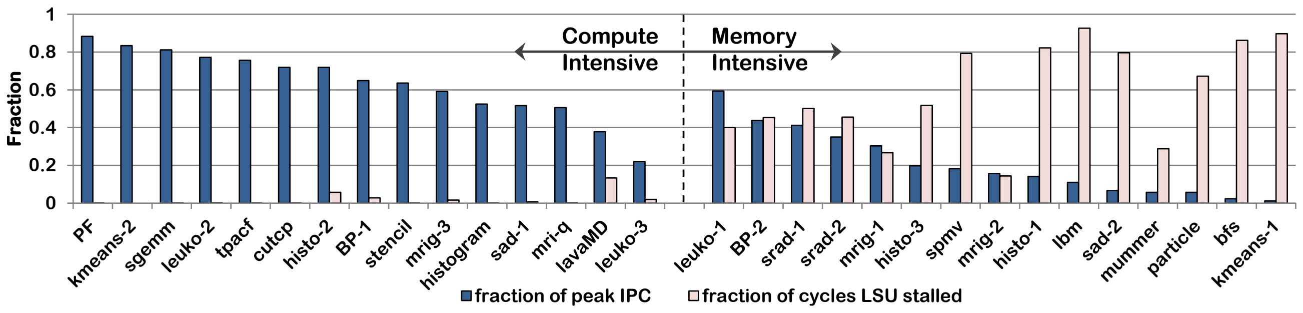

Fig. 2: Fraction of peak IPC achieved and the fraction of cycles for which the LSU stalled due to memory subsystem saturation.

workloads. When examining the impact of memory subsystem back pressure on these applications, we observed that compute intensive kernels rarely stall the LSU. This results in the timely processing of memory requests required to initiate their computation. On the other hand, the memory intensive kernels show a significant rise in the number of LSU stall cycles due to growing back pressure in the memory subsystem. This stalling hampers the GPUs ability to overlap memory accesses with computation. The percent of LSU stall cycles seen in Figure 2 for the memory intensive workloads is indicative that that these workloads struggle to achieve peak throughput primarily due to saturation of the memory subsystem.

C. *Impact of Scheduling on Saturation*

Warp scheduling can have a significant impact on how many memory accesses can be overlapped with computation. Figure 3 shows execution timelines for an example workload with three warps run on three architectures. For the sake of simplicity, in this example we assume that each arithmetic instruction takes one cycle, load instruction takes five cycles and that there are three warps that can execute in parallel. In this miniature system, only one warp can issue an arithmetic or memory operation per cycle. The example workload is shown on the left in the figure. In this workload, the first compute operation occurs on line 3 and it cannot begin until both of the loads finish execution. The top timeline illustrates theoretical GPU hardware that has infinite bandwidth and MSHRs, and uses a round-robin warp scheduler. The second timeline shows the execution for a system with support for two outstanding memory requests and also uses round-robin scheduling. The final timeline demonstrates how Mascar's scheduling works with hardware that supports two outstanding requests.

Infinite resources: When the system has infinite resources with round-robin scheduling, the load for r1 is launched for each of the three warps over three consecutive cycles. At $t = 4$, the load unit is ready for the next load instruction, so the load for r2 gets issued for all three warps in a similar fashion. After five cycles, the load for r1 for W_0 returns, and because there were enough MSHRs and the memory requests were fully pipelined, the loads for r1 for W_1 and W_2 complete in the next two cycles. At $t = 9$, the load for r2 returns and the computation can finally begin for W_0 in the next cycle. This computation is completed by $t = 13$. As only one warp's instruction can be executed per cycle, the computation for all three warps takes 12 cycles for the workload's four add instructions due to round-robin scheduling. This theoretical system finishes execution in 21 cycles for this synthetic workload.

Round-robin with two outstanding requests: The performance of this system is hindered due to it only supporting a limited number of outstanding memory requests. The first two cycles behave similar to the infinite resource case, but in the third cycle, the memory request cannot reserve an MSHR and thus has to wait to be sent to memory until one of the first two requests returns. As the first request comes back at $t = 6$, W_2's load for r1 must be issued at $t = 7$. At $t = 8$, W_0's load for r2 is issued, delaying computation until this load completes at $t = 14$. This computation can hide the memory access of W_1 and W_2's loads of r2. These loads of r2 return one after another at $t = 18$ and $t = 19$. The computation for both of the warps takes 10 cycles to complete as only one warp can execute per cycle in round-robin fashion. The overlap of memory accesses with computation is shown by the bands for memory and compute, respectively. It can be seen that the compute operations are ineffective in hiding any memory accesses until $t = 13$ as none of the data required by any warp is available. The execution of this workload finishes at $t = 26$, five cycles later than the theoretical system with infinite MSHRs.

Mascar with two outstanding requests: Whenever Mascar detects memory subsystem saturation, rather than selecting instructions from warps in a round-robin fashion, it prioritizes memory requests of a single warp. In the illustrated example, the workload has been running for some time such that saturation occurs at $t = 1$. Mascar detects this situation and prioritizes memory requests made by W_0 to be issued at $t = 1$ and $t = 2$. No other memory requests can be sent as the system can only handle two outstanding requests at a time. When the data returns for the first load at $t = 6$, W_1 is given priority to issue its memory requests for load of r1 in the next cycle. At $t = 7$, W_0's memory request for the load for r2 returns and the warp is now ready for execution. W_0's computation can begin simultaneously with W_1's next memory request at $t = 8$. This computation from cycles 8 to 11 completely overlaps with memory accesses, which was not possible when using the round-robin scheduler in either of the previously mentioned architectures. Similarly, the computation for W_1 begins at $t = 14$ and overlaps with memory accesses until it completes at $t = 17$. The program finishes execution at $t = 23$, and due to the increased overlap of computation with memory access, the workload running on Mascar finishes earlier than the traditional scheduler.

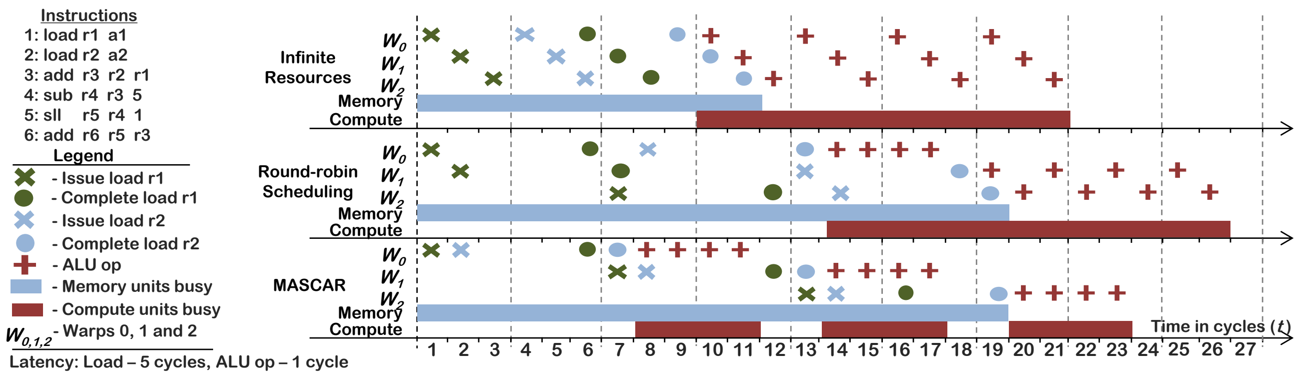

Fig. 3: Execution timeline for three systems. The top illustrates an infinite bandwidth and MSHR system. The middle shows the baseline system with round-robin scheduling. The timeline on the bottom depicts Mascar's scheduling.

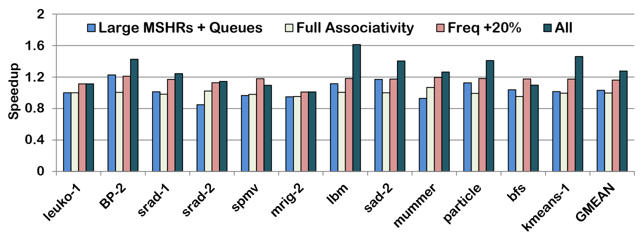

Fig. 4: Performance impact on memory intensive kernels when increasing the number of MSHRs and memory subsystem queues, fully associative L1 and L2 caches, increasing the DRAM frequency by 20%, and a combination of all three.

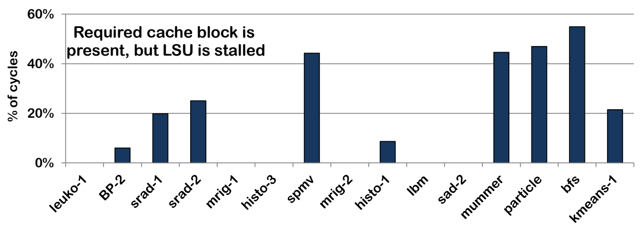

Fig. 5: Fraction of cycles for which there is at least one warp which has one cache block present in the cache that cannot be accessed due to a stalled LSU.

D. Increasing Memory Resources to Resolve Saturation

The most intuitive way to resolve memory resource saturation is to provision the memory subsystem with more resources to handle such demands. To model this, we ran memory intensive kernels on a simulator modeling a NVIDIA Fermi GTX 480 GPU modified to have *a*) a large number of L1 and L2 MSHRs and large queues throughout the subsystem (all sized at 10240); *b*) fully associative L1 and L2 caches; *c*) increasing the memory frequency by 20%; and *d*) a combination of *(a)*, *(b)*, and *(c)*. Figure 4 shows the performance improvements achieved over an unmodified GTX 480 when using these extra provisions. While a combination of all these improvements yields decent speedups for some workloads and modest results for others (Geo-mean of 33% speedup), such a system is extraordinarily hard to realize. This system will have high complexity, energy consumption and is dependent on future technology. As provisioning GPUs with additional or faster memory resources is prohibitive, we elect to focus on better overlapping memory accesses with computation through a new warp scheduling scheme and minimal hardware additions.

E. Impact of Memory Saturation on Data Reuse

The stalling of the LSU during memory saturation also has an impact on the ability of the warps to reuse data present in the cache. As the LSU is stalled due to the back pressure from memory subsystem, warps whose data might be present in the data cache cannot access it. Figure 5 shows the percentage of cycles when the core could not issue any instruction for execution, while the data required by at least one of the warps was present in the cache. It shows that seven of the fifteen kernels have at least one warp's data present in the cache for 20% of the time during saturation. These warps do not need any memory resources as their data is already present in the data cache if a mechanism exists to allow these warps to exploit such hits-under-misses, there is a significant opportunity to improve performance for these kernels through cache reuse.

III. Mascar

To ameliorate the dual problems of slowed workload progress and the loss of data cache locality during periods of memory subsystem back pressure, Mascar introduces a new, bimodal warp scheduling scheme along with a cache access re-execution system. This section details these contributions, and Section IV evaluates their efficacy.

To reduce the effects of back pressure on memory intensive workloads by improving the overlap of compute and memory operations, Mascar proposes two modes of scheduling between warps. The first mode, called the *Equal Priority (EP)* mode, is used when the memory subsystem is not saturated. In this case, Mascar follows the SM's traditional warp scheduling scheme where all warps are given equal priority to access the memory resources. However, if the memory subsystem is experiencing heavy back pressure, the scheduler will switch to a *Memory Access Priority (MP)* mode where one warp is given priority to issue all of its memory requests before another warp can issue any of its requests.

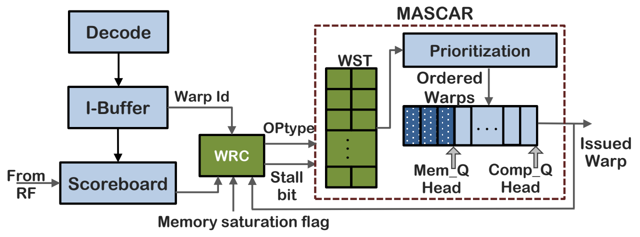

Fig. 6: Block diagram of the Mascar scheduler. The WRC and WST are new hardware structures that interact with the front-end of the SM's pipeline.

The goal of giving one warp the exclusive ability to issue its requests when scheduling in MP mode is to schedule warps to better overlap computation and memory accesses. By doing so, Mascar is able to reduce the impact of performance bottlenecks caused by saturation of resources in the memory subsystem. As discussed in Section II-C, round-robin scheduling permits all warps to issue *some* of their memory requests, but none can continue with their execution until *all* of their requests are filled. This also holds true for state of the art schedulers, e.g. GTO. To prevent this from occurring, Mascar's MP mode gives one warp priority to issue all of its requests while other warps must wait. This makes all of the prioritized warp's data available for computation sooner than in conventional scheduling approaches. As this warp can now compute on its data, another warp can be given priority to access memory, thus increasing the likelihood that computational resources are used even as many other warps wait on their data.

The detection of memory subsystem saturation is accomplished by a signal from the SM's L1 cache, called the memory saturation flag. The details of the logic used to determine the flag's value are explained later in this section. Once this back pressure signal is asserted, the SM switches scheduling from EP to MP mode. If the back pressure is relieved over time, the saturation flag will be cleared and Mascar switches the scheduler back to EP mode, allowing for rapid and simple toggling between Mascar's two modes of scheduling.

To exploit data locality that might be present during periods of memory saturation, Mascar couples a re-execution queue with the LSU. By providing a means for warps to access the L1 data cache while other memory resources are saturated, this queue allows an SM to exploit hit-under-miss opportunities as warps with accesses that might hit in the L1 can run ahead of other stalled accesses. If the access misses in the L1 the system's memory resources have saturated such that the request cannot reserve an MSHR, the request is pushed onto the re-execution queue and its access will be retried at a later time. This reduces the delay a warp incurs between accessing data present in the L1 and when the warp can start computation. Section III-C provides implementation details and an example of Mascar's cache access re-execution in action.

A. *Scheduling in Memory Access Priority (MP) Mode*

In the baseline SM, after an instruction is decoded it is put into an instruction buffer. Once the instruction is at the head of this buffer and its operands are ready, the scheduler will add the instruction's warp to a queue of ready warps according to its scheduling policy. Mascar alters this step by gathering these warps into separate memory-ready and compute-ready warp queues as shown in Figure 6. This allows Mascar to give priority to one memory-ready warp to issue to the LSU and generate its memory requests while stalling all other warps waiting on this resource.

Identifying the memory warp to issue: To track which memory-ready warp should be issued to the LSU, Mascar uses a Warp Status Table (WST) that stores two bits of information per warp. The first bit indicates whether a warp's next instruction will access memory, and the second tells the scheduler to stall issuing of the warp's instruction.

The state of a warp's bits in the WST are determined by the Warp Readiness Checker (WRC). To set a WST entry's memory operation bit, the WRC simply examines the instruction buffer to determine whether or not each warp's next instruction will access memory and sets the bit accordingly. To set a stall bit in the WST, the WRC must first determine which warp is given exclusive ownership and access to the LSU. This warp is called the *owner* warp, and the details of managing ownership are described in *owner warp management*. If a warp's next instruction needs to access memory but it is not the owner warp, the WRC sets its stall bit in the WST. A warp's stall bit will also be set if the scoreboard indicates that an operand needed by that warp's memory or compute instruction is not ready. If a new warp is granted ownership, the stall bits are updated according to the aforementioned process. If during execution the memory back pressure is relieved and the scheduler switches from MP back to EP mode, all stall bits are cleared.

Owner warp management: The owner warp continues to issue all of its memory requests through the LSU as memory resources become available. It does so until it reaches an instruction which is dependent on one of the issued loads. At this point it relinquishes its ownership. In order to identify when an operation is dependent on a long latency load, each scoreboard entry is augmented with one extra bit of metadata to indicate that its output register is the result of such an instruction. The WRC, shown in Figure 6, requests this dependence information from the scoreboard for each instruction belonging to the owner warp, and the scoreboard finds the disjunction of all this new metadata for this instruction's operands. When the WRC is informed that the owner warp's instructions are now waiting on its own loads, the WRC relieves this warp of ownership and resets all other warps' stall bits in the WST. Now that all warps are free to issue to memory, one will go ahead and access memory. If the memory saturation flag remains asserted and the scheduler remains in MP mode, this warp will become the new owner.

Warp prioritization: Mascar prioritizes warps into two groups. The first group is for warps that are ready to issue to the arithmetic pipeline and are called compute-ready warps. Conversely, the second group of warps are called memory-ready warps, and are warps which are ready to be issued to the memory pipeline. These groups are illustrated by the unshaded and shaded regions, respectively, of the ordered warps queue shown in Figure 6. When scheduling in MP mode, compute-ready warps are given priority over memory-ready warps to allow a maximum overlap of computation with memory accesses during periods of heavy back pressure in the memory subsystem. Within these groups, the oldest warp will be scheduled for issue to their respective pipelines.

Once Mascar switches from EP to MP mode, warps that do not have ownership status will no longer be able to issue memory instructions to the LSU. However, earlier instructions from such warps might already be present in the memory-ready warps queue. If Mascar does not allow these warps to issue, the owner warp's memory instructions will not be able to reach the head of the queue, preventing forward progress. To address this potential bottleneck, Mascar allows these non-owner, memory-ready warps to issue to the L1 data cache. If a non-owner's request hits in the L1, its data returns, and the instruction can commit. Otherwise, the L1 will not allow this non-owner's request to travel to the L2, and instead returns a negative acknowledgement. Mascar informs the L1 which warp has ownership, allowing the L1 to differentiate between requests from owner and non-owner warps. Negative acknowledgements may still cause the LSU to stall when non-owner warps get stuck waiting for data to return, but Mascar overcomes this limitation with cache access re-execution, described in Section III-C.

Multiple schedulers: Mascar's scheduling in MP mode allows one warp to issue its memory accesses at a time, but modern NVIDIA GPU architectures like Fermi and Kepler have multiple warps schedulers per SM and are capable of issuing multiple warps' instructions per cycle. To ensure that each scheduler does not issue memory accesses from different warps when MP mode is in use, the WRC shares the owner warp's information with all schedulers present in an SM. Now, the scheduler that is handling an owner warp's instructions will have priority to issue its memory instructions to the LSU during periods of memory subsystem saturation, while any scheduler is free to issue any warp's computation instructions to their respective functional units.

Memory subsystem saturation detection: The memory saturation flag informs Mascar's scheduler of memory back pressure. This flag is controlled by logic in the SM's L1 cache.

The L1 cache has a fixed number of MSHRs as well as entries in the miss queue to send outstanding request across the interconnect. If either structure is totally occupied, no new request can be accepted by the cache that needs to send an outstanding request. Therefore, whenever these structures are almost full, the L1 cache signals to the LSU that the memory subsystem is saturating. The LSU forwards this flag to Mascar's scheduler so that it toggles to MP mode. The cache does not wait for these structures to completely fill as once this occurs, the owner warp will not be able to issue any requests. Instrumenting the L1 cache with this saturation detection is ideal as the L1 is located within the SM. Doing the same at the L2 requires information to travel between the L2 partitions and the SM, likely through the interconnect, which will incur more design complexity and delay Mascar's scheduler from shifting to MP mode in a timely manner.

Our benchmarks show that global memory accesses are the dominant cause of back pressure. However, for certain workloads, texture or constant memory accesses are major contributors to saturation. Mascar also observes saturation from these caches. In all kernels we observed, only one of the three memory spaces causes saturation at a given time.

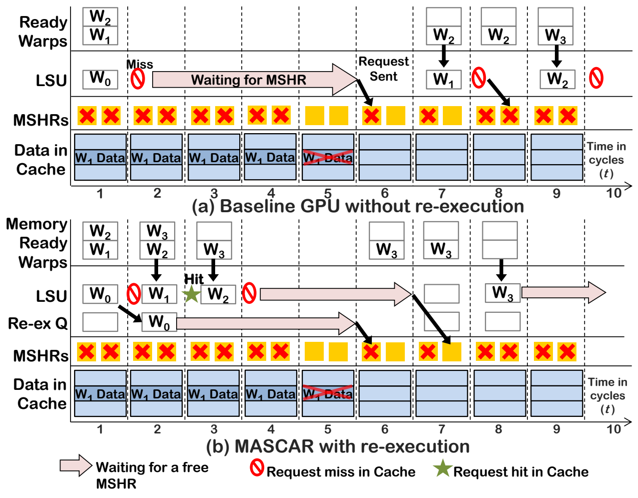

Fig. 7: (a) The baseline system is ready to issue W_0's request but is stalled in the LSU as the L1's MSHRs are full, preventing W_1's request from hitting in the L1. (b) Mascar's cache access re-execution allows W_1's request to hit in the L1 while W_0 waits for an MSHR.

B. Scheduling in Equal Priority (EP) Mode

In contrast to MP mode, in EP mode, the workload is balanced in such a way that that the memory subsystem is not saturated. Therefore, EP mode sends out as many memory requests as possible to utilize the under-utilized memory subsystem by prioritizing warps from memory queue over warps from compute queue. Within each queue, the warps are ordered in greedy-then-oldest policy. Memory warps are prioritized over compute warps because in EP mode the memory pipeline is not saturated and there should be enough computation to hide the latency of all the pipelined memory requests.

C. Cache Access Re-execution

When the memory subsystem becomes saturated with requests, the L1 data cache stops accepting new requests from the SM's LSU. At this point, the LSU is stalled and cannot process any new requests. When a memory request returns from memory and an MSHR is freed, the LSU can issue a new request to the L1. During this time, another warp whose data may be available in this cache cannot progress with its computation as the LSU is stalled. If this warp was able to access the L1 and retrieve its data, it could have helped hide the latency of other memory accesses with its own computation.

An example of this situation is illustrated in Figure 7(a) for the theoretical device depicted in Figure 3 with round-robin scheduling and a load instruction latency of five cycles if the data is not present in the cache. While W_0 is stalled in the LSU as no MSHRs are available in the L1, W_1 and W_2 are ready to access memory but cannot be issued. During this stall, W_1's data is actually available in the cache, but at $t = 5$ this data gets evicted when a previous request completes. After W_0's request gets sent to global memory, W_1 misses in the L1 and must reload its data from the L2/global memory. If there was some mechanism in place to enable a hit under miss while the MSHRs were occupied, W_1 could have gone ahead with its memory request to the cache, accessed the required data and started computation, all without needing to reload its data.

When the LSU stalls, warps with memory requests that might hit in the L1 cache are served their data much later. If this delay is too long, there is a chance that what may have been a hit in the L1 will become a miss as another request might return from the L2/DRAM and evict this request's data. This effect is exacerbated due to the limited size of the data cache, for which 1536 threads share up to 48KB of L1 data cache in modern architectures. Also, a system with more MSHRs might provides warps with more opportunities to access the cache, however, having more MSHRs can also exacerbate cache thrashing. Without increasing the size of an already small L1 cache, adding support for more outstanding requests may force some of these requests to evict data that would have soon been reused.

To overcome these issues and take advantage of hits under misses, we propose to add a cache access re-execution queue alongside the LSU as shown in Figure 8. The queue enables hit under miss without sending new requests to the L2/DRAM for non-owner warps. Whenever a request stalls in the LSU, the generated address and associated meta-data is removed from the head of the LSU's pipeline and is pushed onto the re-execution queue, freeing the LSU to process another request. If the newly processed request misses in the L1 cache, it is also added to this queue. Otherwise, if the next request hits in the cache, that warp can commit its memory access instruction and continue execution.

Requests queued for re-execution are processed if one of two conditions are met. First, if the LSU is not stalled and has no new requests to process, it can pop a request from the re-execution queue and send it to the L1. Second, if the re-execution queue is full, the LSU is forced to stall as it cannot push more blocking requests onto this queue. If this occurs, the LSU only issues memory requests from its re-execution queue. New memory instructions can only be issued to the LSU once entries in the re-execution queue are freed and the LSU is relieved of stalls. Address calculation need not be repeated for queued accesses, as this was already done when the request was first processed by the LSU.

Mascar only allows one memory instruction per warp to be pushed to the re-execution queue at a time. This is to ensure that if a store is followed by a load instruction to the same address, they are serviced in sequential order and thereby maintaining the weak memory consistency semantics of GPUs. As an instruction from a warp may generate several However, our results show that a 32 entry re-execution queue satisfies the memory demands of our benchmarks' kernels. This design incurs less overhead than adding 32 MSHRs per SM as the complexity of a larger associative MSHR table is avoided.

The impact of coupling a re-execution queue with the LSU is illustrated in Figure 7(b). To demonstrate a simple example, this queue only has one entry. At $t = 0$, W_0 gets issued to the LSU, and because the MSHRs are full it is moved to the re-execution queue at $t = 2$. Now, W_1 is issued to the LSU, before a prior request evicts its data from the cache as had occurred in Figure 7(a). By moving W_0's request to the re-execution queue, W_1 can now go ahead and access the L1, where it experiences a hit under W_0's miss. Having obtained its data, W_1's memory instruction can commit, allowing W_2 to issue to the LSU. As the re-execution queue is full and W_2 misses in the L1 at $t = 4$, the LSU is forced to stall. However, as W_1 finished its load, the SM is able to perform useful computation on that data while W_0 and W_2 await an MSHR. Furthermore, as W_1's memory instruction was serviced and hit earlier, better utilization of the MSHRs is possible, allowing W_3 to be issued to the LSU earlier than in the baseline system. By exploiting hit under miss opportunities when using a re-execution queue, warps are able to bypass other warps that normally would block access to the L1, permitting more reuse of data in the cache.

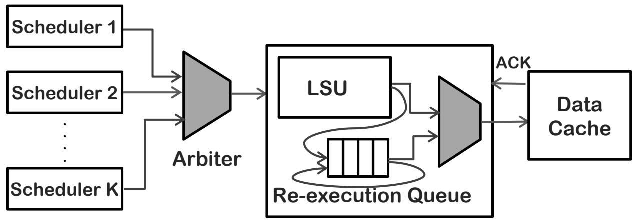

Fig. 8: Coupling of the re-execution queue with an LSU. When a request from the LSU is stalled it is pushed to the re-execution queue. The cache sends an ACK if it accepts a request, or a NACK otherwise.

TABLE I: Simulation Parameters

Architecture	Fermi (GTX 480, 15 SMs, 32 PEs/SM)
Schedulers	Loose round-robin (LRR), Greedy-then-oldest (GTO) OWL [15], CCWS [27], Mascar
L1 cache/SM	32kB, 64 Sets, 4 way, 128 B/Line, 64 MSHRs [23]
L2 cache	768kB, 8 way, 6 partitions 64 MSHRs/partition, 200 core cycles latency
DRAM	32 requests/partition, 440 core cycles latency

TABLE II: New Hardware Units per SM

Unit	Entry Size	# Entries	Total
Warp Readiness Checker	6 bits	n/a	1 byte
Warp Status Table	2 bits	48	12 bytes
Re-execution Queue	301 bits	32	1216 bytes

D. MP Mode and Re-execution

The impact of re-execution on a memory subsystem experiencing back pressure is important. We have to ensure that re-execution does not interfere with MP mode scheduling as re-execution can send requests to the L2/global memory for any warp, not just the owner. Mascar resolves this at the L1 by preventing misses from non-owner warps from accessing the next level cache. As described in Section III-A, Mascar provides the L1 with knowledge of which warp holds ownership when MP mode is active. If a non-owner warp's request is sent from the queue to the L1 and misses, the L1 returns a negative acknowledgement and this request is moved from the queue's head to its tail. Otherwise, if this request belonged to an owner warp and enough resources were available to send the request across the interconnect to the L2/DRAM, it would send this request to get its data. If an owner warp's request missed in the L1 and could not be sent to the L2/DRAM, the request would also be moved to the re-execution queue's tail. This recycling of requests, as shown

TABLE III: Benchmark Descriptions

Application	Type	Comp/Mem	Application	Type	Comp/Mem
backprop (BP-1)	C	107	mrig-1	M	13.4
backprop (BP-2)	M	19	mrig-2	M	22.4
bfs	M	2.4	mrig-3	C	147
cutcp	C	949	mri-q	C	3479
histo-1	M	15.3	mummer	M	4.9
histo-2	C	35.9	particle	M	3.5
histo-3	M	14.3	pathfinder (PF)	C	55.83
histogram	C	91.2	sgemm	C	75
kmeans-1	M	0.27	spmv	M	4.2
kmeans-2	C	88	sad-1	C	2563
lavaMD	C	512	sad-2	M	10.2
lbm	M	10.4	srad-1	M	25
leuko-1	M	28	srad-2	M	22
leuko-2	C	173	stencil	C	38
leuko-3	C	5289	tpacf	C	10816

in Figure 8 ensures that the owner warp can make forward progress when its requests are in the middle of the queue. If a warp relinquishes ownership and the scheduler's memory-ready warps queue is empty, the warp of the request at the head of the re-execution queue is given ownership and can now send requests to the L2/DRAM.

Data Reuse Improvement: Mascar improves L1 data cache hit rates in two ways. First, the owner warp alone can bring new data to the L1 and thereby the data brought by different warps is reduced significantly. Rogers et al. [27] have shown that for cache sensitive kernels, intra-warp locality is more important than inter-warp locality. Therefore, it is important for warps to consume the data they request before it gets evicted from the cache. By prioritizing one warp's requests over others, Mascar allows one warp to bring its data to the L1 and perform computation upon it before it is evicted by another warp's data. Second, re-execution also enables reuse for warps whose data has been brought to the cache by enabling hit-under-miss, leading to an overall improvement in hit rate.

IV. Experimental Evaluation

A. Methodology

We use GPGPU-Sim [3] v3.2.2 to model the baseline NVIDIA Fermi architecture (GTX 480) and the Mascar extensions. We use the default simulator parameters for the GTX 480 architecture and the relevant parameters are shown in Table I. All of our benchmarks, shown in Table III, come from the Rodinia [5], [6] and Parboil [33] benchmark suites. The last column in Table III shows the number of instructions executed by the SM per miss in the L1 cache. Benchmarks that exhibit a ratio of instructions executed per L1 miss of greater than 30 are considered compute intensive and are marked C in the type column, and others are marked M for memory intensive. GPUWattch [20] is used to estimate the power consumed by these applications for both the baseline and Mascar.

We compare Mascar scheduling with Loose round-robin (LRR, our baseline), Greedy-then-oldest (GTO), OWL [15] and Cache Conscious Wavefront Scheduling (CCWS) [27]. CCWS is modeled using [26] with simulator parameters modified to match our baseline GTX 480 architecture.

B. Hardware overhead

Three hardware structures are added to the baseline SM to implement Mascar's modifications. Table II shows the per SM overheads of these structures. To support scheduling requirements for MP mode, the Warp Status Table (WST) stores two status bits for each warp. As the Fermi architecture supports a maximum of 48 warps per SM, the WST requires 12 bytes of storage. The Warp Readiness Checker (WRC) stores the current owner warp's ID in a six bit field, and uses simple, single-bit boolean logic to determine the stall bit.

To support Mascar's cache access re-execution, each re-execution queue entry stores 301 bits of information. This includes the request's base address (64 bits), each thread's byte offset into this segment (224 bits — 7 bit 128B segment offset $\times$ 32 threads), and bits to identify the warp (6 bits) this request belongs to and its destination register (7 bits). A sensitivity study in Section IV-C found that 32 entries were sufficient to expose ample hit under miss opportunities, making the queue 1216 bytes in size. Comparing the queue to each SM's 64KB L1 data cache/shared memory using CACTI 5.3 [34], we find that the queue's size is just 2.2% of that of the cache and that a queue access uses 3% of the energy of an L1 access which is a small component of the overall energy consumption.

C. Results

Performance Improvement: Figure 9 shows the speedup achieved for memory intensive kernels when using the warp scheduling schemes mentioned in Section IV-A with respect to a round-robin scheduler. The chart is splits memory intensive kernels into those that are cache sensitive and those that are not. GTO focuses on issuing instructions from the oldest warp, permitting this one warp to make more requests and exploit more intra-warp data reuse. This greedy prioritization allows GTO alone to achieve a geometric mean speedup of 4% for memory intensive and 24% for cache sensitive kernels. However, GTO swaps warps whenever the executing warp stalls for the results of long latency operations including floating point operations, barrier synchronization and lack of instruction in instruction buffer. This allows memory accesses to be issued by more than one warp in parallel during memory saturation, resulting in the phenomenon explained in Section II-C. The overall speedup achieved by GTO over the baseline is 13%.

OWL [15] tries to reduce cache contention by prioritizing sub-groups of warps to access the cache in an attempt to give high-priority sub-groups a greater chance to reuse their data. OWL is effective for several workloads that are sensitive to this prioritization, such as BP-2, mrig-1, histo-3 and particle. Overall, however, OWL is not as effective as reported for the older GTX 280 architecture due to improvements in our Fermi baseline. Prior work has shown that preserving inter-warp data locality is more beneficial when improving the hit rate of the L1 data cache [27]. The scoreboarding used in modern architectures allows a warp to reach an instruction reusing cached data much faster, enabling higher reuse of data in the cache. Overall, OWL scheduling shows 4% performance improvement over the baseline. We do not implement OWL's memory-side prefetching as it is orthogonal to our work and is applicable for any scheduling scheme.

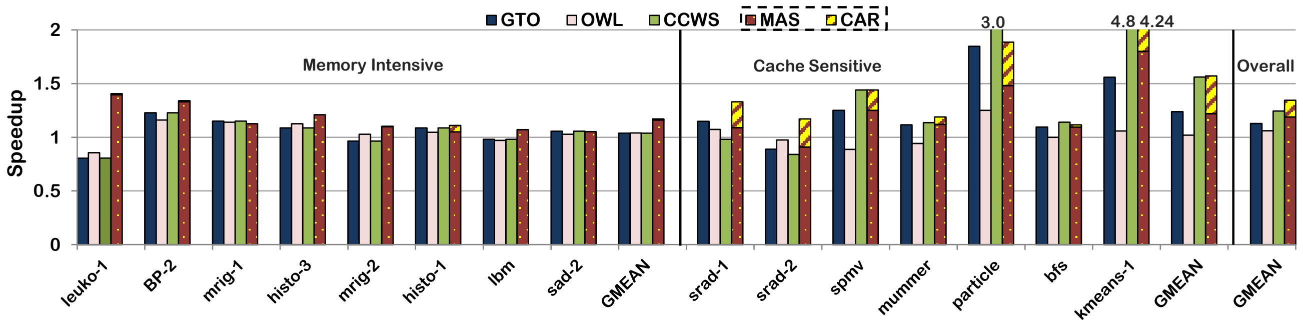

Fig. 9: Comparison of performance for different policies on memory intensive kernels, relative to round-robin scheduling. The two components for the last bar in each kernel represents the contribution of scheduling and cache re-execution, respectively.

For cache insensitive workloads, CCWS is equivalent to GTO, achieving a speedup of 4%. CCWS shows a significant speedup of 55% over the baseline for cache sensitive kernels as it is designed specifically to improve performance of such kernels with reuse, leading to an overall speedup of 24%.

The speedup achieved by Mascar is due to two components: memory aware scheduling and cache access re-execution (stacked on top of each other in Figure 9 for the last bar). For memory intensive kernels, Mascar performs better than or almost equal to all other scheduling schemes except for mrig-1 and mrig-2. These kernels have brief phases of high memory intensity, and the memory back pressure in these phases is relieved before these benchmarks can benefit from Mascar. While CCWS provides 4% speedup for memory intensive kernels, Mascar provides a significantly higher speedup of 17%. This is because CCWS only focuses on cache sensitive kernels, whereas Mascar improves the overlap of compute and memory for all kinds of memory intensive kernels.

The main reason for the high speedup of Mascar over GTO is that GTO provides priority to the oldest warp, whereas Mascar provides exclusivity to one warp. This inherently leads to higher utilization in cases where there is data locality. For example, leuko-1's innermost iteration reads two float data from the texture cache. The first data is constant across all warps in a thread block for that iteration, and the second is brought in by one warp and can be used by one other warp in the same block. However, with GTO, all other warps will bring their own data for the second load. This results in the eviction of the first warp's data before the reuse by the other warp can occur. With Mascar, all data from one warp is brought in and only then can another warp bring in its data. In the meantime, the data brought in by the first warp can be reused by the second warp. Kmeans-1 is a very cache sensitive kernel. Each warp brings in a significant number of cache lines per load due to uncoalesced accesses. Several of these loads are also reused by another warp. Therefore, thread throttling schemes such as CCWS do very well on it. Similarly, Mascar slows down the impact of the heavy multi-threading by allowing only one warp to issue requests at a time and its results are close to CCWS for kmeans-1. With GTO, more cache thrashing takes place due to inter-warp cache thrashing between 48 warps.

Cache access re-execution alone provides performance benefits of 35% and its combination with Mascar scheduling provides an overall speedup of 56% for cache sensitive kernels.

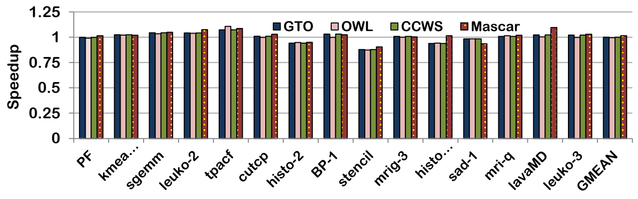

Fig. 10: Performance of compute intensive kernels for four different scheduling policies. Overall, the choice of scheduler has little impact on these workloads.

CCWS is designed specifically for cache sensitive kernels and provides a speedup of 55%. While Mascar's speedup for cache sensitive workloads is comparable to CCWS, the area overhead of Mascar is only one-eighth of CCWS' additional hardware. The design complexity of CCWS involves a victim tag array and a point-based system for warps, whereas the design of Mascar involves a queue and tables such as WST and WRC which do not use an associative lookup. Overall, Mascar achieves a geometric mean speedup of 34% as compared to 13% speedup for GTO and 24% for CCWS at one-eighth the hardware overhead.

The choice of scheduling policy has little impact on the performance of compute intensive workloads, as seen in Figure 10. OWL, GTO and CCWS are within 1% of the baseline's performance, while Mascar achieves a 1.5% geometric mean speedup. Occasionally, short phases in compute intensive workloads suffer from memory saturation, and Mascar can accelerate these sections. Leuko-2, lavaMD, and histogram are examples of such workloads. Leuko-2's initial phase exhibits significant pressure on the memory system due to texture accesses, which Mascar's MP mode alleviates. LavaMD and histogram behave similarly but with saturation due to global memory accesses. Mascar does cause performance degradation compared to the baseline scheduling for histo-2, stencil and sad-1, but these slowdowns are not significant.

Microarchitectural impacts of Mascar: Mascar's scheduling and re-execution have a significant impact on the fraction of cycles for which the LSU stalls, previously discussed in Section II-B. Figure 11 shows that Mascar is capable of reducing these stalls on average by almost half, from 40% down to 20%. By alleviating these stalls, Mascar more efficiently brings data to the SM and overlaps accesses

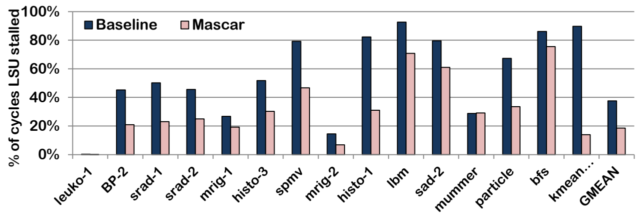

Fig. 11: Reduction of LSU stalls when using Mascar.

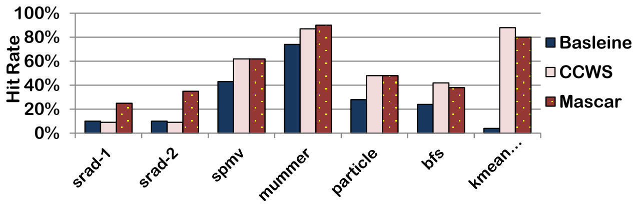

Fig. 13: Improvement in L1 data cache hit rates.

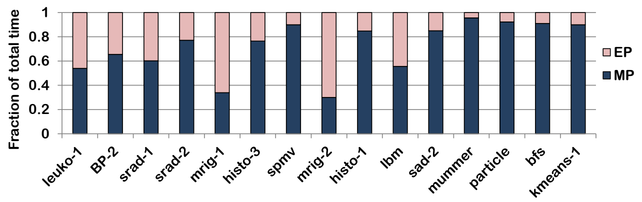

Fig. 12: Distribution of execution time in EP and MP modes across workloads.

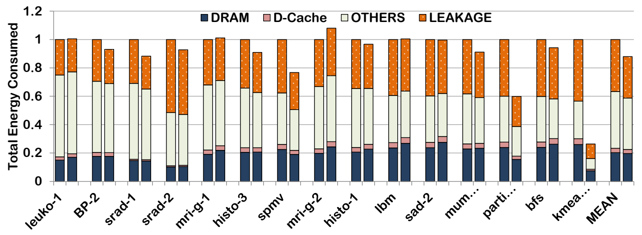

Fig. 14: Energy consumption breakdown for baseline (left bar) and Mascar (right bar, relative to baseline).

with computation. Spmv, particle and kmeans-1 experience great reductions in stalls, which correlates with their significant speedups in Figure 9. Because compute intensive workloads stall the LSU far less frequently than memory intensive applications, we do not include these figures.

Depending on the severity of memory intensity, Mascar's scheduler will be in either EP or MP modes for different durations of time as shown in Figure 12. There is a direct correlation between the number of cycles a workload is in MP mode to the number of cycles the LSU is stalled for the baseline scheduler used in Figure 2. Leuko-1, srad-1, mrig-1, mrig-2 and lbm are workloads that spend some time in both modes and exhibit phased behavior with intermixed periods of compute and memory intensity. As previously described, mrig-1 and mrig-2 spend most of their time in EP mode and their MP mode phases are so short that the benefits of Mascar are muted for these benchmarks. Severely memory intensive workloads, including kmeans-1, bfs, particle, and mummer, operate in MP mode for 85%–90% of their execution.

The impact of the re-execution queue is illustrated by the improvements in L1 hit rates shown in Figure 13. We only show results for the seven cache sensitive workloads and compare them with CCWS. While CCWS achieved better hit rates than Mascar for three kernels as it reduces the number of warps that can access the data cache to preserve locality, Mascar's hit rate improvements are better than CCWS for three other kernels. Because the number of requests going to DRAM are significantly reduced, the exposed memory latency is reduced such that it better overlaps with computation. The resulting exposed latency can be effectively hidden by both CCWS and Mascar, resulting in less than 1% difference in performance between the two. However, CCWS's victim tag array incurs more hardware overhead and design complexity than Mascar, which has a that can induce a higher energy overhead while not improving the performance of cache insensitive kernels.

We performed a sensitivity study of the impact of the size of the re-execution queue on performance of the cache sensitive kernels. For all kernels except kmeans-1, a 16 entry re-execution queue is sufficient to expose greater reuse. Kmeans-1 has a significant number of uncoalesced accesses, and requires more entries as each uncoalesced access breaks down into multiple requests. Overall, the performance of the kernels saturates with 32 entries, and hence was chosen for our design.

Improvement in Energy Efficiency: Mascar's speedups from scheduling and re-execution leads to energy savings. This can be seen in Figure 14, where each workload's left and right bar represent the energy consumed by the baseline versus Mascar architectures. On average, Mascar reduces energy consumption by 12% compared to the baseline system. To further analyze Mascar's energy efficiency, we break these energy figures down into four components: *DRAM*, *L1 data cache*, *leakage* and *others*, which includes the interconnect, SM pipelines, and L2 cache. All components are normalized with respect to the total energy consumed on the baseline.

DRAM energy consumption is mostly unchanged except for kmeans-1, particle, and spmv show noticeable reductions. By reducing the warps that can issue memory requests during MP mode scheduling, less thrashing and higher L1 hit rates reduce DRAM traffic. A few kernels (srad-2, mrig-2, and lbm) experience increased DRAM energy consumption. Jog et al. [15] discuss how consecutive thread blocks access the same DRAM row. In the case of these kernels, Mascar sometimes reduces row locality by allowing all of one warp's requests to go to DRAM, forcing other warps to reopen previously used rows. The data cache's energy consumption is slightly reduced as Mascar exploits hit-under-miss opportunities, reducing failed access attempts. This indicates that the energy impact of re-executing cache accesses is negligible. Other components exhibit a 3.5% decrease in energy, primarily due to reduced interconnect traffic in MP mode. The greatest contribution to energy savings is due to the savings in leakage energy, which improved by 7% on average. As workloads made quicker progress during phases of memory saturation, they finished earlier, thus reducing the leakage energy consumed.

V. Related Work

Warp scheduling: Scheduling on GPUs has received wide attention from many works. Jog et al. [15] propose a scheduler which aims to reduce cache contention and exploit bank level parallelism to improve performance. With the addition of scoreboarding, newer GPUs have improved abilities to overlap computation with memory accesses and alleviate cache contention. Scoreboarding allows warps to reach loads faster, hence taking advantage of available data locality. Narasiman et al. [22] proposed a two-level warp scheduler (TLS) which divides warps into fetch groups to overlap computation with memory access by staggering warp execution so that some warps perform computation while others execute memory operations, modern GPU architectures, however, allow enough warps per SM to naturally generate an equivalent or better overlapping. Our experiments show that TLS scheduling shows 1.5% speedup over the baseline.

Improving cache locality in GPUs: Rogers et al. [27] propose a cache conscious scheduling mechanism which detects cache locality with new hardware and moderates the warps that access the cache to reduce thrashing. DAWS [28] improves upon this mechanism by using a profile-based and online detection mechanism. Jia et al. [13] use a memory request prioritization buffer (MRPB) to reorder cache requests so that the access order preserves more data locality. Furthermore, in MRPB, certain requests bypass the cache and are transferred and received by the core directly. Cache and Concurrency Allocation (CCA) [35] also limits the number of warps that can allocate cache lines to improve data locality while other warps bypass the cache, ensuring sufficient bandwidth utilization.

Mascar targets general memory intensive workloads that may not have data locality, which is not addressed by CCWS, DAWS, MRPB or CCA. While these schemes try to avoid stalling the LSU during back pressure, Mascar enables the LSU to continue issuing requests to the L1 even when it is stalled through its re-execution queue. It also improves data reuse by only allowing one warp to bring its data, preventing other warps from thrashing the cache. Mascar incurs less hardware overhead than the above techniques. We compare Mascar with OWL and CCWS in Section IV-C.

Improving resource utilization on GPUs: There have been several works on improving GPU resource utilization. Gebhart et al. [9] propose a unified scratchpad, register file, and data cache to better distribute the on-chip resources depending on the application. However, they do not provide resources to increase the number of outstanding memory requests. Jog et al. [14] propose a scheduling scheme that enables better data prefetching and bank level parallelism. Rhu et al. [25] create a locality aware memory hierarchy for GPUs. Lakshminarayana and Kim et al. [18] evaluate scheduling techniques for DRAM optimization and propose a scheduler which is fair to all warps when no hardware-managed cache is present. Our work focuses on systems with a saturated memory system in the caches and DRAM. Ausavarungnirun et al. [2] propose a memory controller design that batches requests to the same row to improve row locality, hence improving DRAM performance.

Adriaens et al. [1] propose spatial multi-tasking on GPUs to share resources for domain specific applications. Gebhart et al. [8] propose a two-level scheduling technique to improve register file energy efficiency. Kayiran et al. [16] reduce memory subsystem saturation by throttling the number of CTAs that are active on an SM. Fung et al. [7] and Meng et al. [21] optimize the organization of warps when threads diverge at a branch. Bauer et al. [4] propose a software-managed approach to overlap computation and memory accesses. In this scheme, some warps bring data into shared memory for consumption by computation warps. Hormati et al. [11] use a high-level abstraction to launch helper threads that bring data from global to shared memory and better overlap computation.

Re-execution: The concept of re-executing instructions has been studied for CPUs by CFP and iCFP [32], [10] which insert instructions into a slice buffer whenever the CPU pipeline is blocked by a load instruction and re-execute from a checkpoint when a miss returns. Sarangi et al. [30] minimize this checkpointing overhead by speculatively executing forward slices. While these schemes benefit single threaded workloads, having slice buffers and checkpointing micro-architectural state for every warp can be prohibitively expensive. Hyeran et. al propose WarpedDMR [12] to re-execute instructions on the GPU during under-utilization of the hardware, providing redundancy for error protection. They focus on re-execution of instructions that run on the SM and do not re-execute accesses to the data cache, which is done by Mascar to improve performance.

VI. Conclusion

Due to the mismatch between the applications' requirements and resources provided by GPUs, the memory subsystem is saturated with outstanding requests which impairs performance. We show that with a novel scheduling scheme that prioritizes the memory requests of one warp rather than issuing several requests of all the warps, better overlap of memory and computation can be achieved for memory intensive workloads. With a re-execution queue, which enables access to data in the cache while the cache is blocked from accessing main memory, there is significant improvement in cache hit rate. Mascar achieves 34% performance improvement over the baseline scheduler. Due to the low overhead of Mascar, this translates to an energy savings of 12%. For compute intensive workloads, Mascar is 1.5% faster than the baseline scheduler.

ACKNOWLEDGEMENTS

This work has been was supported by ARM Ltd. and National Science Foundation grant SHF-1217917. The authors would like to thank John Kloosterman, Shruti Padmanabha, other members of the CCCP research group, and the anonymous reviewers for their insightful suggestions and feedback.

References

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012.

[2] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 416–427, 2012.

[3] A. Bakhoda, G. L. 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 Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009.

[4] M. Bauer, H. Cook, and B. Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[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 IEEE Symposium on Workload Characterization*, pages 44–54, 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. pages 1–11, 2010.

[7] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007.

[8] 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 *Proc. of the 38th Annual International Symposium on Computer Architecture*, pages 235–246, 2011.

[9] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. Proc. of the 45th Annual International Symposium on Microarchitecture, pages 96–106, 2012.

[10] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating all-level cache misses in in-order processors. *IEEE Micro*, 30(1):12–19, 2010.

[11] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–392, 2011.

[12] J. Hyeran and M. Annavaram. Warped-dmr: Light-weight error detection for gpgpu. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 37–47, 2012.

[13] W. Jia, K. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 272–283, 2014.

[14] 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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 332–343, 2013.

[15] A. Jog, O. Kayiran, 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. 21th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 395–406, 2013.

[16] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, 2013.

[17] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010.

[18] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. Dram scheduling policy for gpgpu architectures based on a potential function. *Computer Architecture Letters*, 11(2):33–36, 2012.

[19] J. Lee and H. Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. Proc. of the 18th International Symposium on High-Performance Computer Architecture, pages 1–12, 2012.

[20] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: enabling energy optimizations in gpgpus. In *Proc. of the 40th Annual International Symposium on Computer Architecture*, pages 487–498, 2013.

[21] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 235–246, 2010.

[22] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317, 2011.

[23] C. Nugteren, G. J. van den Braak, H. Corporaal, and H. Bal. A detailed gpu cache model based on reuse distance theory. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 37–48, 2014.

[24] NVIDIA. *CUDA C Programming Guide*, May 2011.

[25] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A locality-aware memory hierarchy for energy-efficient gpu architectures. pages 86–98, 2013.

[26] T. G. Rogers. CCWS Simulator.
https://www.ece.ubc.ca/ tgrogers/ccws.html.

[27] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. Proc. of the 45th Annual International Symposium on Microarchitecture, pages 72–83, 2012.

[28] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 99–110, 2013.

[29] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[30] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. Reslice: selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, 2005.

[31] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 440–451, 2012.

[32] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Ghandi, and M. Upton. Continual flow pipelines: achieving resource-efficient latency tolerance. *IEEE Micro*, 24(6):62–73, 2004.

[33] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, 2012.

[34] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, Hewlett-Packard Laboratories, Apr. 2008.

[35] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive cache and concurrency allocation on gpgpus. *Computer Architecture Letters*, PP(99), 2014.