Managing DRAM Latency Divergence in Irregular GPGPU Applications

Niladrish Chatterjee*[†][§], Mike O'Connor[†][‖][§], Gabriel H. Loh[‡], Nuwan Jayasena[‡] and Rajeev Balasubramonian*

*University of Utah †NVIDIA ‡Advanced Micro Devices, Inc. (AMD Research) ‖University of Texas at Austin

Abstract—Memory controllers in modern GPUs aggressively reorder requests for high bandwidth usage, often interleaving requests from different warps. This leads to high variance in the latency of different requests issued by the threads of a warp. Since a warp in a SIMT architecture can proceed only when all of its memory requests are returned by memory, such latency divergence causes significant slowdown when running irregular GPGPU applications. To solve this issue, we propose memory scheduling mechanisms that avoid inter-warp interference in the DRAM system to reduce the average memory stall latency experienced by warps. We further reduce latency divergence through mechanisms that coordinate scheduling decisions across multiple independent memory channels. Finally we show that carefully orchestrating the memory scheduling policy can achieve low average latency for warps, without compromising bandwidth utilization. Our combined scheme yields a 10.1% performance improvement for irregular GPGPU workloads relative to a throughput-optimized GPU memory controller.

I. INTRODUCTION

The energy efficiency of Graphics Processing Units (GPUs) [28] and the development of high-level parallel programming models such as CUDA [39] and OpenCL [29] have led to the increasing adoption of the GPU for running data-parallel workloads. The most recent energy-efficient supercomputers all rely heavily on general purpose GPUs (GPGPUs) to scale up their parallel and floating point throughput [19]. Efforts to scale GPU performance and energy-efficiency are critical to enabling the next-generation of Exascale supercomputers [5].

The data-parallel, SIMD nature of GPU architectures have traditionally been optimized for dense, highly-structured workloads common in graphics applications. There has been considerable effort in recent years, however, to develop GPU implementations of a wide variety of parallel algorithms from the High Performance Computing (HPC) and the enterprise domains. Many of these applications are irregular and exhibit control flow and memory access patterns that are not readily amenable to the GPU's architecture [12], [21], [34], [36]. In particular, many of these new irregular applications demonstrate significant *Memory Access Irregularity (MAI)* [11] that leads to performance bottlenecks [9]. The memory accesses in these programs are often data dependent, and they tend to have less spatial locality compared to traditional graphics and regular general-purpose GPU applications.

One source of MAI-induced performance degradation is the Single-Instruction, Multiple-Thread (SIMT) execution model of GPUs. In the GPU, each SIMT core runs a group of threads (a warp) in lockstep. When a warp issues a load instruction, the warp will block once an instruction dependent

upon the load data becomes the next to issue. This warp is unable to make progress until *all* the data for the constituent load instructions is available. Given the SIMT architecture, a single delinquent load can block forward progress of the warp. This introduces the problem of *memory latency divergence* – a warp can be stalled until the last memory request from a vector load instruction is returned, potentially long after other memory requests from the vector load have completed. In many workloads, this load latency cannot be hidden by executing other warps. Several studies have highlighted how memory latency divergence can be a significant performance bottleneck in GPUs [35], [45]. This problem of memory latency divergence is not unique to GPUs and can also manifest itself in other SIMD/vector architectures that support "gather" load operations (e.g. [2]).

Memory latency divergence arises from several factors. First, the cache hierarchy can service different requests at different times due to hits and misses at various levels. Second, current GPU memory controllers are primarily optimized to support traditional, structured workloads with low degrees of MAI. These modern GPU memory controllers will extensively re-order incoming requests to maximize memory system bandwidth with no explicit policy to manage the relative service time of different requests from the same warp. The out-of-order service can often delay a subset of the requests from one warp while memory requests for other warps (and other GPU functions) are serviced.

In this paper, we focus on reducing the latency divergence arising from DRAM accesses due to the memory controller's scheduling decisions. We focus on the main memory system because it is the source of the most significant portion of memory latency divergence. While cache hits can cause some requests to be serviced with relatively low latency, the concurrent execution of thousands of threads in a GPU causes caches to have poor hit-rates. Consequently, several requests for a warp will often be serviced at the memory controllers. Through an opportunity analysis we show that if a system could eliminate *all* main memory latency divergence, then overall GPU performance improves by 43%. Consequently, the crux of our proposals focus on enabling the memory scheduler to ensure that different requests from a warp are serviced together, or in quick succession, without interference from other warps. To achieve this objective, we propose handling the requests from a warp loosely as a single group (called a *warp-group*) at the memory controller. We then use a novel prioritization scheme between different warp-groups to reduce the average memory-induced stall time for all warps (Section IV-B). We also observe that a significant fraction of warps issue requests to multiple memory channels. In a baseline GPU, each channel operates completely independently and thus requests issued by the same warp encounter different latencies at different controllers. We show that if the memory controllers are able

---

to exchange limited information to coordinate their scheduling decisions, we can achieve further reductions in latency divergence (Section IV-C). In essence, by exchanging information between GPU cores and memory controllers, we are enabling *warp-aware memory controllers*.

While reducing latency divergence is important for irregular GPGPU applications, bandwidth is still a first-order concern for applications both with and without MAI. We address this issue in two ways. First, the prioritization function described in Section IV-B ensures that when warps exhibit row-locality at the DRAM (as in regular GPGPU compute applications and graphics applications), the proposed scheduler works at least as well as a bandwidth-optimized baseline. Second, for applications with MAI, we relax the requirement of servicing *all* of a warp's requests in perfect succession. We formulate a policy (Section IV-D) that achieves high bandwidth utilization and reduced average warp completion times by overlapping row-hit requests from one warp with row-miss requests from other nearly-complete warps. Finally, we augment our warp-aware scheduler to also be aware of upcoming write drain periods in the memory system (Section IV-E). Our combined policies achieve a 10.1% performance improvement over a throughput optimized baseline, and a 7.3% improvement over a previously proposed SIMT-aware memory scheduler.

II. BACKGROUND

In this section, we take a brief look at the typical architecture of a modern GPU and its DRAM scheduler.

A. GPU Cores

A GPU consists of a number of compute cores (Streaming Multiprocessor or SM in NVIDIA parlance) each of which is assigned one or more groups of warps to execute. Each SM consists of multiple SIMD lanes (we model 32 lanes) and threads in a warp are run on these lanes in lockstep. On a load instruction, each lane generates a memory request and the warp is blocked till all the requests are serviced by the memory.

B. Memory System

The SMs have private L1 caches and are connected to different memory partitions through a crossbar interconnect. Each memory partition consists of a slice of the shared L2 cache and a GDDR5 channel (we model 6 memory channels for our studies). Each GDDR5 [23] channel is typically 64-bits wide with the command and address bus running at 1.5 GHz. The data bus runs at 3 GHz and is DDR. Each GDDR5 chip has 16 banks and we consider two x32 GDDR5 devices per channel that are operated in tandem as one rank. The GDDR5 chip architecture is specialized for high bandwidth. It has higher bank counts, the banks are organized into bank-groups that accommodate lower bank-conflict delays between different bank groups, a higher I/O frequency, and a more robust power delivery network that allows more frequent row-activations compared to DDR3 (i.e., enabling a lower t_{FAW}). We model all of these aspects in our simulations (Section V).
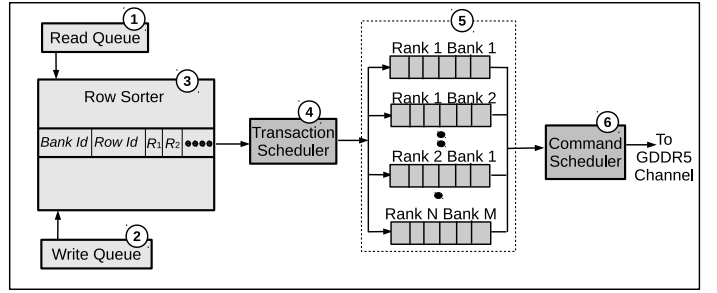

Fig. 1. Baseline GPU Memory Controller

C. Baseline Memory Controller Organization

Memory controllers in throughput processors are optimized to provide high bandwidth. Fig. 1 shows the components of a typical GPU memory controller (GMC) that we model as the baseline for our studies.

The **Read Queue** (1) and the **Write Queue** (2) buffer the read and write requests received from the interconnect along with associated information such as address, requester (SM id), and arrival timestamp. As requests arrive, they are sorted by their bank and row addresses, and entered into the **Row Sorter** (3). The row sorter may have (say) eight entries per bank, representing pending requests to different rows in that bank. Each new request is merged with an existing entry (to form a stream of row-hit requests) or creates a new entry in the row sorter. The **Transaction Scheduler** (4) picks requests from the row-sorter and enqueues the commands required to complete the request in the corresponding **Command Queue** (5).

The transaction scheduler picks a row-hit stream from the row sorter to service in each bank and interleaves requests to different banks, thus exploiting row-buffer locality and bank-level parallelism. To limit latency, the transaction scheduler also attempts to prevent the starvation of older row-miss requests by using an age-based prioritization threshold, as well as a maximum row-hit streak limit.

Write requests to DRAM are the result of write-backs from the last level cache and are not on the critical path for program execution. Writes are buffered in the write queue and are drained when the write queue occupancy rises above a high water mark or when there are no requests in the read queue [14], [48]. The write requests are typically drained until the write queue occupancy reaches a low water mark. This mechanism ensures that the bus does not frequently alternate between reads and writes, thus reducing the DRAM bus turnaround penalties ($tWTR$).

The **Command queues** are maintained on a per-bank basis (since most GPUs only have a single rank on a channel) and store the DRAM commands (e.g., ACT, PRE, COL_RD, COL_WR) for transactions that have been scheduled by the transaction scheduler. The **Command Scheduler** (6) is responsible for issuing the DRAM commands to the GDDR5 chips. The command scheduler enforces low-level command timing restrictions and tracks the state of the different DRAM banks. It iterates over the different queues to interleave requests to different ranks/banks so as to leverage bank-level parallelism. However, within a bank, it issues commands in queue order to
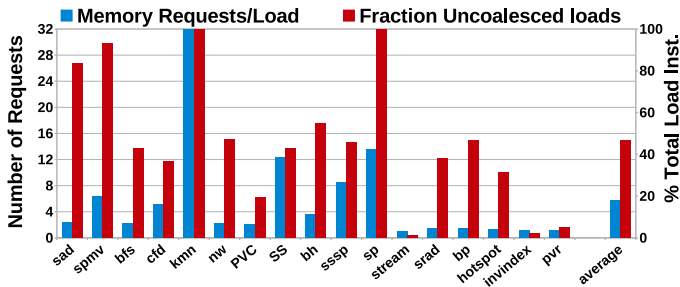
Fig. 2. Coalescing efficiency.

avoid disrupting the scheduling decisions taken by the transaction scheduler. The bank-group architecture of GDDR5 has the advantage of lower inter-command delays when commands are issued to different bank groups [23]. The command scheduler thus tries to interleave requests between different bank groups first and then within each bank group through a multi-level round-robin policy.

In GPUs, the address mapping policy is designed to harvest spatial locality in the access stream and high parallelism across channels, ranks, and banks. First, consecutive cache lines are mapped to the same row in the same bank to promote row buffer locality. To exploit bank- and channel-level parallelism, blocks of consecutive cache-lines are interleaved across the memory channels and banks at a granularity of 256 bytes. In addition, to prevent pathological channel camping, where unusual access strides lead to excessive contention on one or few of the six simulated channels, the channel address is formed by XOR-ing a subset of higher-order bits with some lower order bits to provide a better spread across the channels as follows:

```
channel address =
    {addr[47:11]:(addr[10:8] XOR addr[13:11])} % 6
```

Similarly, to prevent strided accesses from camping on the same bank, the bank address is formed by XOR-ing the bank address bits with a portion of higher-order address bits from the cache set index [53].

III. MOTIVATION

A. Architectural Influences on Memory Latency Divergence

In this section we look at the architectural features in GPUs that affect memory latency divergence (besides the obvious SIMT execution model) and those that try to mitigate latency divergence.

Memory Coalescing: A common architectural technique that reduces memory bandwidth demand in SIMD architectures is *memory coalescing*. In this technique, the individual requests from a warp are combined, based on their target address, to form as few cache-line sized (128B) requests as possible. Coalescing is primarily designed to reduce bandwidth requirements by eliminating redundant accesses to the same cache line. However, it also reduces the opportunity for memory latency divergence by minimizing the number of distinct memory requests per warp. Coalescing is not effective, however, if the data touched by the threads in a warp are not spatially colocated, as is commonly the case in irregular applications. Based on our simulations, we found that the coalescer is

extremely effective for traditional compute workloads from the CUDA SDK [39], but its effectiveness falls short for workloads with irregular memory accesses. Fig. 2 shows that 56% of loads (the black bar) issued by irregular programs result in more than one memory request and that on average each load generates 5.9 memory requests after coalescing (benchmarks and evaluation techniques explained in Section V). This shows that innovations beyond the coalescer are needed to handle memory divergence for irregular applications.

GPU memory schedulers: A warp's requests arrive at the memory controller within short intervals of each other. So, at a first glance, it might seem that a scheduler that processes requests in arrival order would have the effect of servicing a warp's requests (a warp-group) together. However, in practice, the requests from different SMs arrive at the memory controller through the interconnect after L1 and L2 lookups. Thus, memory accesses from different warps intermingle at the read queue and this interleaving renders a simple First-Come, First-Serve (FCFS) policy ineffective. Also, while the transaction scheduler may process a warp-group in a roughly FCFS manner, the individual requests will get placed in per-bank queues that may have different occupancies. As a result, the warp's requests do not complete within short intervals of each other under the FCFS policy. Further, a naive FCFS policy leads to extremely poor bandwidth utilization. A simple First-Ready First-Come-First-Served (FR-FCFS [42]) or the state-of-the-art GMC scheduler will be much more bandwidth efficient, but will aggressively re-order requests to maximize requests to open DRAM rows. If all the requests from a warp are targeted to the same row in the same bank, a FR-FCFS or GMC policy will naturally yield most of the benefits of a warp-aware policy. We observed, however, that warps in irregular GPGPU programs often issue requests to different rows, banks, and channels of DRAM. On average, a warp touches 2 banks and out of its requests, only 30% belong to the same DRAM row. This was also observed by Lakshminarayana et al. in a previous study [32]. These request characteristics require an effective warp-aware scheduler to consider bank occupancies, memory intensity of warps, and write queue drain policies to reduce latency divergence and simultaneously maintain good bandwidth utilization for regular compute and graphics workloads.

Multithreading: GPU SIMD cores leverage thread-level parallelism to hide memory access latency. The effect of long divergence-induced stalls can be mitigated if there are enough ready warps in the system to hide the latency of the slowest request. Previous studies [18], [27] have shown, however, that in spite of having a large number of thread contexts to choose from, a GPU SIMD core will frequently sit idle as all the warps are stalled on pending memory accesses. For instance, recent NVIDIA GPUs support at most 48 to 64 warps within a compute unit [1], while main memory latencies have been measured to exceed 400 cycles [22]. Thus, it is clear thread-level parallelism cannot always completely hide main memory access latency [7].

Caches: Average memory latency can be improved with caching. Requests from a warp that hit in a cache can be returned sooner. Furthermore, the corresponding reduction of traffic that would otherwise be serviced at the memory controller allows other requests to be serviced with reduced
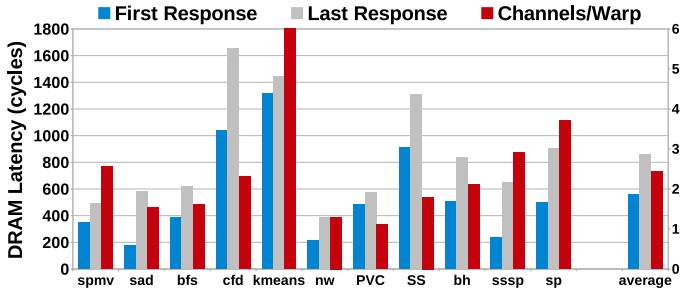
Fig. 3. Extent of memory latency divergence.

Fig. 4. Room for improvement.

queuing latency and contention for DRAM resources. For memory latency divergence to be meaningfully addressed with caches, a substantial fraction of warps must be able to serve all of their memory requests with cache hits. Otherwise, the cache misses for a warp-group will be serviced by the memory controller, and face issues described above. Schemes that exploit better utilization of the cache space [43], [44] may provide synergistic benefits with the memory-controller-based proposals in this work.

B. Quantifying Memory Latency Divergence

The performance penalties associated with memory latency divergence have been documented in recent work [35], [45]. In this section, we further analyze the impact of DRAM latency divergence, particularly on workloads that exhibit MAI.

Main Memory Latency Divergence. To assess the scope of main memory latency divergence, we measure the average gap between the first and last request served by the main memory for each warp (Fig. 3, showing results only for benchmarks that generate more than one memory request on average per load after coalescing). This provides an estimate of the main memory latency divergence in each benchmark. We see that on average, the last request's latency is $1.6\times$ the latency of the first request. This is the latency as seen by the SM and includes arbitration, interconnect, L2 lookup, and queuing delays at the memory controller. The latency of a DRAM request is dependent on the memory controller's scheduling policy. The GMC scheduling policy is optimized to increase throughput and save power. It does not currently service the requests from a single warp together or in quick succession. If a subset of the memory requests belonging to a warp are de-prioritized by a scheduler (e.g., because they caused row-buffer misses), then the warp's progress is hampered. A warp with a high number of row-buffer hit requests can unduly stall a warp with fewer, but low-locality requests. Also, interleaving requests from two different warps can increase the stall-time of the last request from *both* warps. This increases the average effective memory latency experienced by both warps. If the requests from one of the warps could finish in close succession with little interference from the other, then the overall average memory-related stall time for the system would be reduced. The situation is made worse if requests from a warp are sent to different memory controllers. The memory controllers take scheduling decisions independent of each other based on their local load and request characteristics. Fig. 3 shows that each warp touches 2.5 memory controllers on average and motivates the need for coordination between the memory controllers to
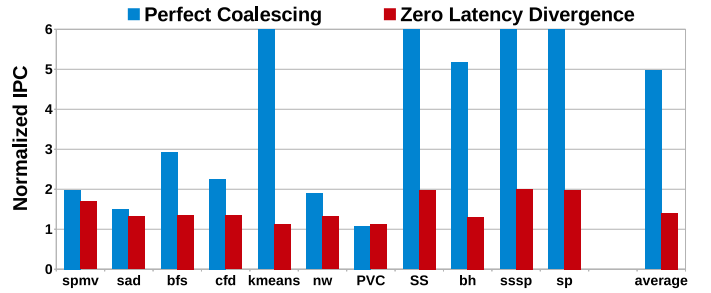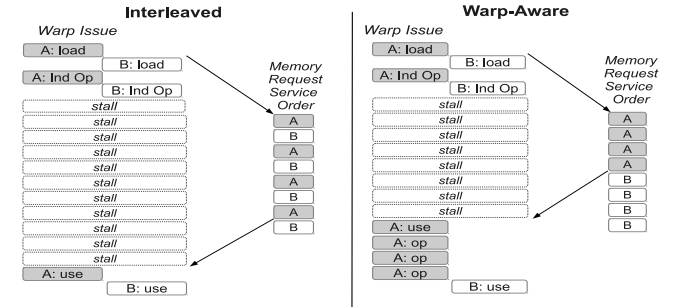

Fig. 5. Avoiding Inter-Warp Interference to Reduce Average Effective Memory Stall Time

reduce latency divergence.

Performance Impact of Memory Latency Divergence. To estimate the impact of memory latency variation, we look at two hypothetical systems. In Fig. 4, the first bar (*Perfect Coalescing*) shows the performance improvement that could be achieved if every warp generated exactly one memory access per load instruction. This leads to a $5\times$ speedup over the baseline system. This is obviously an unrealizable system. The second bar (*Zero Latency Divergence*) shows the performance improvement that can be obtained if all memory requests from a warp could be returned to the SM in close succession without any gaps between them after the first request has been serviced - in essence, if there was no main memory latency divergence. The 43% improvement demonstrated by this experiment represents the upper bound of the improvement that can be obtained by eliminating memory latency divergence in GPUs. It is important to note that this model abstracts away the bank conflicts for all but one request for each warp, but still faithfully models DRAM bus bandwidth and contention. Thus, while a real-world warp-aware main memory system is unlikely to achieve these ideal benefits, the results are encouraging as they indicate significant opportunity for warp-aware management of DRAM.

IV. WARP-AWARE MEMORY SCHEDULING

A. Key Idea

The key idea behind the warp-aware scheduling schemes is illustrated in Fig. 5. It shows two warps A and B issuing N requests each to the memory system, with each request requiring a T-cycle service time from memory. If the requests are processed in an interleaved manner, then the final requests for warps A and B are returned at cycle (2N-1)*T and 2N*T, respectively, leading to an average stall time of $(2N-\frac{1}{2})*T$

cycles. On the other hand, if warp A could get all its requests back before B, then the average memory stall time is 1.5N*T cycles. Our warp-aware scheduling scheme tries to achieve this effect by reducing inter-warp interference and services requests from a warp as close together in time as possible. At a high level, the proposed scheduling policy attempts the following:

- Return all of a warp's requests in close succession. This will also require the schedulers in different channels to coordinate their scheduling decisions.
- Favor shorter warps to reduce the average warp-completion time, possibly at the cost of increased latency for some other requests.
- Maintain high bandwidth utilization and low overall memory latency by exploiting row hits and bank-level parallelism.

To achieve this, we first make a fundamental change in the GMC memory controller. As discussed in Section II, the GMC's row-sorter creates streams of row-hit requests. Each stream might consist of requests from different warps. Instead, we form batches of requests from a single warp and each such group is called a **warp-group**. Instead of selecting a row-hit stream to service, the transaction scheduler picks a completed warp-group and schedules its requests before scheduling requests from other warp groups. To aid the transaction scheduler, priorities are assigned to different warp-groups based on the controller's occupancy, and the state of the DRAM banks. The priorities are dynamically updated when the state changes in response to the scheduling of new requests by the controller. In addition, we update the priority of a warp-group when requests from the same warp are serviced in a different memory controller.

B. Warp-Aware Scheduling: Single Controller

First we discuss a warp-group scheduling scheme that bases decisions solely on the information available in a single controller (referred to as **WG**). In essence, this is a shortest-job-first (SJF) scheduler that arbitrates between the different warp-groups of memory requests with the aim of minimizing the average service time across all warp-groups. The service time for a warp-group is the latency of the last-served request for that warp. It is well-known that SJF can reduce the average wait time of enqueued tasks, but in a DRAM system, true SJF can only be achieved by being cognizant of the state of the DRAM system. Simply considering the number of requests in a warp-group to determine the shortest job is not adequate. Due to locality and load on the different banks accessed by a warp, a warp-group with few row-miss requests may have a long service time, and stall a warp-group with more requests, all of which are row-hits. Also, only one row-miss request from a warp may be pending on a bank with many pending row-hits, even though all of the warp's other requests have been returned from memory. The priority scheme in WG accounts for the locality and bank-level parallelism of the requests in the group (besides the total number of requests), the state of the DRAM banks and bank groups, and the occupancy of each of the bank-level command queues. The scoring system effectively calculates the total service time of each completely formed warp group. The warp-group with the lowest score is prioritized for servicing and the requests from this warp-group are serviced together.
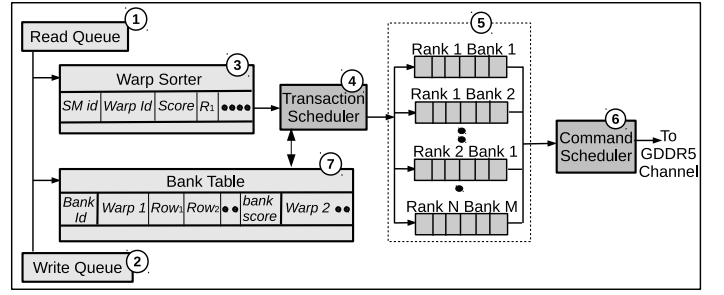

Fig. 6. Warp-Aware Memory Controller

1) Scheduling Policy: The "smarts" of the WG scheduler is in the technique used to rank warp-groups for scheduling. The scores assigned to each warp-group reflect the completion times of the warp-groups and are thus inversely related to the warp-group's priority. The WG transaction scheduler looks for a new warp to schedule after the current warp's requests have been sent to the command queues and picks the warp-group with the smallest score (the shortest job). In the case of a tie, the warp with the highest number of row-hits is picked because row-hits help minimize DRAM power consumption.

Determining the score of the warp-group requires estimating the completion latency of the warp-group based on the type of requests in the warp-group (i.e., row hit/miss), the bank-level parallelism of the requests in the warp-group, and also the state of the DRAM banks (number of queued requests, and active row address). The final score of the warp-group is the maximum score of its requests.

To determine the score of a particular request, it is important to know if it will be a hit or miss in the bank when it is finally scheduled, i.e., whether the last request scheduled in that bank has a matching row-address. If the request is a hit, we assign a score of 1 to it, and if it is a miss, a score of 3 is assigned. The rationale for this is that servicing a row-miss incurs a delay of 36 ns ($t_{RP}+t_{RCD}+t_{CAS}$) compared to 12ns (t_{CAS}) for a row-hit. After assigning the DRAM array access latency score to a warp-group's request, we add a queuing latency score. This is determined by adding the total score of all the requests pending in the corresponding bank's command queue. The scores for a warp-group are updated when new requests are added, and also when a request is scheduled to a bank's command queue.

2) Implementation Details: Fig. 6 shows the microarchitecture of the WG controller. Relative to the baseline in Fig. 1, we see that the Row Sorter structure has been replaced by a Warp Sorter and Bank Table. The Row Sorter in the baseline may have 128 entries, each representing a different <bank,row> tuple. Similarly, the **Warp Sorter** ③ could be a 128-entry structure, each entry representing a different <SM-id, Warp-id> tuple, i.e., a warp-group. Each entry tracks the different addresses that belong to that warp-group. The **Transaction Scheduler** ⑤ pulls out an entire warp-group and places the requests and their commands in the respective **Command Queues** ⑤. The Transaction Scheduler pulls out a warp-group based on its assigned score. The **Bank Table** ⑦ is used to estimate these scores and is described next.

The Bank Table has an entry per bank. The entry for each

bank tracks the pending warp-groups and memory requests to that bank. For each pending warp-group, a bank score is maintained, which represents the expected delay to drain that warp-group. The score is updated as each request is received from the read/write queues. The score is also updated every time a warp-group is pulled out by the Transaction Scheduler, i.e., each bank score is incremented based on the requests that have just been scheduled to that bank. The Transaction Scheduler reads these scores every cycle. For a given warp-group, the maximum bank score represents the warp-group's completion time. Among the warp-groups that have been fully transferred from the SMs to the GMCs, the Transaction Scheduler picks the warp-group with the smallest score and sends it to the Command Queues. This requires a mechanism to determine when a warp-group has been fully transferred from the SMs – this is done by tagging the last request from a warp-group to a GMC. Note that the interconnect between the SMs and GMCs does not re-order requests from a single SM, even though it can interleave requests from different SMs.

C. Warp-Aware Scheduling: Multiple Memory Controllers

As shown in Fig. 3, in irregular benchmarks, a single warp often generates requests to multiple memory controllers. The single channel warp-aware scheduling scheme (WG) can only ensure that a warp-group is serviced as a unit from a single channel. However, different warp-groups belonging to the same warp will have different completion times across the different controllers. To mitigate this issue, we present a mechanism that tries to reduce the latency divergence across channels with coordinated scheduling across the channels.

Overview. We augment the WG scheme so that the priority of a warp-group in a memory controller is determined not only by the state of its channel, but also by whether requests from the same warp have already received service in other controllers. In this scheme (referred to as **WG-M**), memory controllers exchange information over a dedicated point-to-point interconnect (distinct from the crossbar connecting the memory partitions and SMs). When a warp-group is selected for service at a memory-controller, a coordination message, comprising of the selected warp-id and the score of that warp-group in the memory controller is broadcast to the other five memory controllers. The score represents the expected completion time of the warp-group in the source memory-controller. Every cycle, a controller checks its five receiver ports on the coordination network and matches the received warp-group-ids against the warp-groups in its tracking structures. On a match, the score of the warp-group may be decreased if the current controller is deemed to be delaying the warp's requests significantly. For this, the received remote completion time, represented by the remote score (RC), is compared against the local completion time or score (LC). If RC is greater than LC, no action is taken. If LC is greater, then the local score of the warp-group is decreased by *(LC-RC)* to prioritize the warp-group.

The primary overhead for this design is a narrow all-to-all network. We assume a network that has 30 16-bit links. When a warp-group is selected by a transaction scheduler, a 32-bit message consisting of the SM id, warp id, and local completion time is sent to the other five memory controllers.

Banks	MERB
1	31
2	20
3	10
4	7
5	5
6-16	5

TABLE I. MERB TABLE FOR GDDR5

D. High-Bandwidth Warp-Aware Scheduling

The warp-aware memory scheduling algorithms described so far target warp-group service time, without concern for effects on bandwidth utilization. The prioritization of warp-groups by these schemes can interrupt streaks of row-hit traffic, introducing more frequent row-misses and more DRAM-timing induced idle cycles on the interface. This reduction in effective bandwidth results in additional latency in servicing requests due to increased queuing delays in the memory controller. This increased latency reduces the effectiveness of the inter-controller coordination mechanisms that are attempting to minimize warp-group service latency.

To address this challenge, we augment the WG-M policy with a DRAM request scheduling strategy (called **WG-Bw**) that carefully chooses when a row-miss request can be introduced without adversely impacting the delivered bandwidth. This is achieved by maximizing the overlap of a row-miss access in one bank with row-hit accesses in other banks. Thus, the overheads of precharging and activating in one bank are "hidden" with data transfers from the other banks.

Central to this scheduling scheme is a metric called the Minimum Efficient Row Burst (*MERB*). This metric is simply the number of data transfers required to other DRAM banks in order to hide the costs of a row-miss in a given bank. This metric is a function of DRAM timing parameters and the number of other banks with requests pending to an open row.

To determine the number of requests needed to hide the overheads of a row-miss request, we consider the sequence of commands to transfer data from a given DRAM row. First, an activate command fetches the data from the DRAM bank's arrays to its row buffer. After sufficient time has passed (t_{RCD}), a read command can be issued to read out the data from a portion of the row buffer. The data is delivered over multiple (t_{BURST}) cycles. Several read commands can be issued consecutively to stream out a number of row-hit requests. Finally, a precharge command can be issued only after sufficient time has elapsed since the last activate (t_{RAS}) and the last read (t_{RTP}) commands. This precharge command closes the row so that a subsequent activate can be issued to that bank after a delay (t_{RP}). It can be seen from this sequence of operations that the read-to-precharge delay (t_{RTP}), the precharge-to-activate delay (t_{RP}), and the activate-to-read delay (t_{RCD}), determine the overhead to switch from servicing row-hits in a given bank to handling a row-miss request.

Thus, the *MERB* value when multiple banks have pending work is simply the number of bursts required to cover this overhead divided by the number of other banks with pending work. Note, however, that another pair of DRAM timing

parameters, the activate-to-activate delay (t_{RRD}) and the four-activate-window (t_{FAW}), limit the rate at which activates can rotate among banks. If enough banks have pending requests, the overhead that needs to be hidden is this period between activates among the different banks.

Thus, depending on the number of banks with pending requests, (b), the *MERB* value is:

$$MERB = \begin{cases} max(\frac{tRTP+tRP+tRCD}{(b-1)*tBURST}, \frac{max(tRRD,\frac{tFAW}{4})}{tBURST}) & b > 1 \\ 31 & b = 1 \end{cases}$$

In the case where only a single DRAM bank has pending work, there is no way to hide the overheads of a row-miss. If there are a sufficient number of row-hit requests per activate command, such that $(t_{RCD} + t_{BURST}*n + (t_{RTP}-t_{BURST}+t_{CK})) \geq t_{RAS}$, then the bandwidth utilization is given by:

$$utilization = \frac{tBurst * n}{tRCD + tBurst * n + (tRTP - tBurst + tCK) + tRP}$$

Substituting GDDR5 values, if there are 2 or more read requests per activate to a single bank, the utilization is:

$$utilization = \frac{1.33 * n}{(1.33 * n) + 25.33}$$

Thus, we set the *MERB* value to 31 (the limit of a 5-bit per-bank counter), which delivers up to 62% utilization in the single bank case.

This simple table of *MERB* values, shown for GDDR5 timings in Table I, is indexed by the number of banks with pending traffic, and can be computed at boot-time or loaded from the boot ROM of a given product. These *MERB* values are used by the memory transaction scheduler to decide how many row-hit requests should be scheduled before allowing a row-miss request from an older warp. This simple accounting scheme requires the memory controller to maintain only a small 5-bit counter per DRAM bank and the small pre-computed *MERB* table. The per-bank counters keep track of the number of row-hit requests that have been serviced since the row was activated. When a higher-priority row-miss wants to be serviced on a given bank and some row-hits are pending for that bank, the counter is examined and if it is less than the current *MERB* value (selected from the table given the number of other banks that have work pending), the row-miss is postponed. The additional row-hits are serviced, each incrementing the counter, until the row-hits are exhausted or the *MERB* threshold is met. If the *MERB* threshold is met, the number of remaining row-hit requests is examined. If only one or two row-hit requests remain, then these requests are also serviced before allowing the row-miss request to be serviced. This "orphan control" policy prevents a row-miss request from leaving behind only one or two requests to a row, which itself would likely lead to poor bandwidth utilization.

This *MERB*-based scheduling scheme (WG-Bw), therefore, will try to schedule a number of transfers to each bank so that precharge/activate timings can be hidden. By holding off scheduling a row-miss until a given bank has serviced its minimum efficient row burst of data transfers (plus preventing
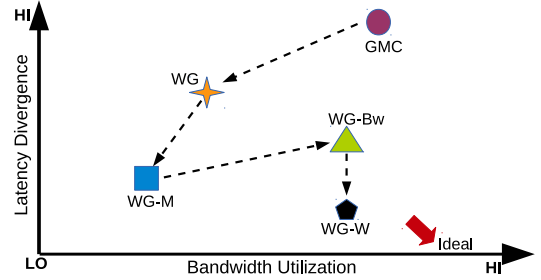

Fig. 7. Latency Divergence and Bandwidth Impact of Schedulers (not to scale)

leaving behind one or two orphan requests), this scheme ensures that, on average, most row-miss overheads will be hidden by data transfers in other banks.

In order to keep bandwidth utilization high, this *MERB*-based scheduling scheme may introduce a bounded additional latency to a higher-priority row-miss to allow up to *MERB*+2 row-hits to be serviced. Since each of these additional row hit requests can be serviced with 2*(t_{CK}) additional latency, the total additional latency is bounded at (*MERB*+2)*2*(t_{CK}). For the case where only 1 bank has work pending this is 44ns with our GDDR5 timings. A more typical case, where there are at least 4 banks with pending requests, this is 12ns. This is a small amount of additional worst-case latency for these row-miss requests, particularly when considering that a typical warp-load request sees ~500ns of total latency. This small incremental latency to one request enables more effective bandwidth utilization, thereby reducing average queuing delays for *all* requests in the memory controller.

E. Warp-Aware Write Draining

As mentioned in Section II, writes are buffered and drained in batches in the DRAM controller to avoid frequent bus-turnarounds (t_{WTR}) [14]. The reads stalled by a write-drain can experience significant queuing delays [14], [50]. We propose the **WG-W** policy, an extension of the WG-Bw policy, that monitors the write-queue drain and changes the read scheduling policy before the write-drain is triggered. WG-W prioritizes warp-groups that have only one request (regardless of their score), once the write-queue occupancy gets within eight entries of its high water mark.

F. Summary

Fig. 7 shows how we navigated the latency divergence *vs.* bandwidth utilization space to develop the warp-aware, bandwidth-optimized scheduling schemes starting from the baseline GMC.

V. METHODOLOGY

We use GPGPU-Sim version 3.1.2 [3], [8], which has been verified against real hardware for a range of different kernels [4], to model a GPU similar to the NVIDIA GTX-480. The salient features of the GPU are listed in Table V. We integrate the DRAM timing model from the USIMM [13] DRAM simulator after modifying USIMM to model GDDR5 timing and the GMC-model described in Section II-C. We model a

GPU System Configuration	
No. of Compute Units	30
Warp Size	32
Max Threads/Core	1024
L1 cache/Core	32KB, 128B cache-line size 8-way assoc. LRU
Number of DRAM channels	6
L2 cache/Memory partition	128KB, 128B line-size 16-way assoc, LRU
DRAM device	Hynix GDDR5 H5GQ1H24AFR [23]
DRAM Configuration	6 64-bit Channels 2 x32 Chips/Channel 16 Banks/Chip 4 Banks/Bank Group
GDDR5 Pin Bandwidth	6.0 Gbps
GDDR5 Clk period (tCK)	.667ns
DRAM Read Queue	64 entries per controller
DRAM Write Queue High/Low Watermarks	64 entries per controller 32/16
GDDR5 Timing Parameters	tRC=40ns, tRCD=12ns, tRP=12ns tCAS=12ns, tRAS=28ns, tRRD=5.5ns tWTR=5ns, tFAW=23ns, tWL=4 tCK tRTP=2ns, tBURST=2 tCK tRTRS=1 tCK tCCDL=3 tCK, tCCDS=2 tCK

TABLE II.   SIMULATION PARAMETERS.

Suite	Benchmarks (abbreviations)
Rodinia	Breadth-First Search(bfs), CFD Solver(cfd) Needle-ManWunsch(nw), Kmeans Clustering(kmeans)
MARS	PageViewCount(PVC), SimilarityScore(SS)
LonestarGPU	Survey Propagation(sp), Barnes-Hut(bh), Single-Source Shortest Paths(sssp)
Parboil	Sparse Matrix Dense-Vector Multiplication(spmv), Sum of Absolute Differences(sad)

TABLE III.   WORKLOADS

Hynix 1Gb GDDR5 DRAM part [23] and the timing constraints that were modeled are listed in Table V.To evaluate our proposals we use benchmarks from Parboil [47], Rodinia [15], Mars [20] and Lonestar [11] suites. Because our proposals are targeted at irregular GPGPU workloads with MAI, we selected those benchmarks that are sensitive to memory system performance and demonstrate memory access divergence (i.e., produce more than a single uncoalesced memory access per load instruction on average). These applications are listed in Table V. While warp-aware scheduling is targeted at these irregular benchmarks, in Section VI-A, we discuss the effect of our proposals on benchmarks that are memory sensitive, but not memory divergent. We run each benchmark for 1 billion instructions or to completion, whichever is earlier.

In GPGPU-Sim, TLB misses, and page-walks are not simulated, because in modern GPUs (which use fairly large page sizes), TLBs have virtually 100% coverage. In the future, if TLB misses become important [41], then our schemes will perform better than the baseline GMC-controller. If a subset of memory requests from a warp are stalled on a TLB miss, then our schedulers will not waste precious DRAM bandwidth to service the other requests for that warp. On the other hand, the MERB scheme can handle the sparse memory requests generated by page-table pointer lookups more efficiently by overlapping them with row-hit streams of other warps.
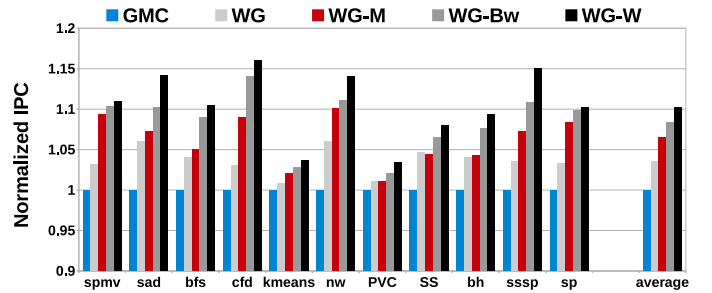

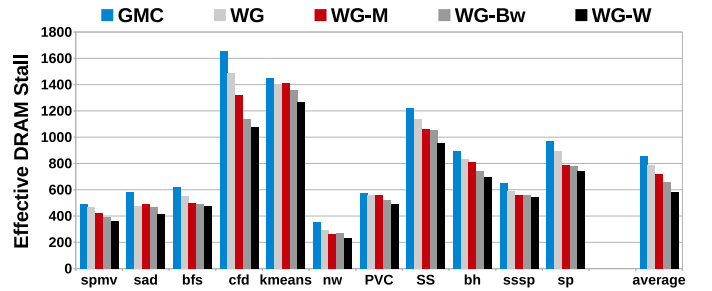Fig. 8.   Performance normalized to the GMC-baseline

Fig. 9.   Effective main memory latency experienced by warps

VI.   EVALUATION

In this section, we analyze the performance impact of the following four scheduling schemes against the baseline GMC.

- Warp-group scheduling per controller (WG)
- Warp-group scheduling across multiple controllers (WG-M)
- Bandwidth optimized warp-group scheduling for multiple controllers (WG-Bw)
- Warp-aware write-drain applied to WG-Bw (WG-W)

We also show the benefits of our proposed schedulers over two other previously proposed GPU memory scheduling schemes [32], [51]. We finally analyze the impact of our schemes on regular compute workloads which show no memory divergence and are typically aided only by throughput optimizations.

Fig. 8 shows the performance improvement in terms of the IPC (normalized to the baseline GMC scheduler). The best performing scheduler (WG-W) obtains a 10.1% improvement in throughput over the GMC-baseline across the irregular applications. The performance benefits from the different scheduling optimizations are largely additive. Fig 9 shows how the average DRAM induced stall-time for warps (i.e., the time to service the last request for the warp) is improved by the different scheduling mechanisms.

The major fraction of the benefit obtained by WG-W comes from reducing or eliminating the latency divergence for a warp across the main memory system. We see that the WG-M scheduler provides a 6.2% average improvement in IPC. When warp-aware scheduling is contained within each individual controller (WG), then the average improvement in performance is limited to 3.4%. The impact of the shortest-job-first policy imbued in the priority scheme of WG and WG-M scheduling is reflected in the reduction of the average effective latency of
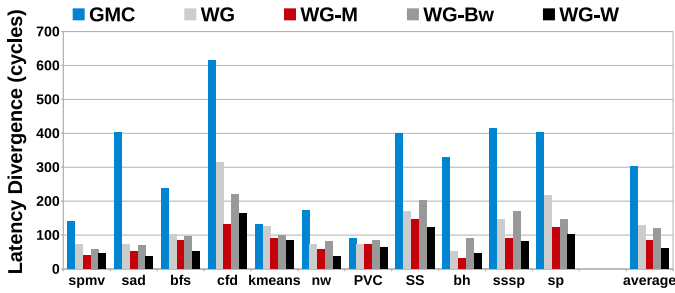
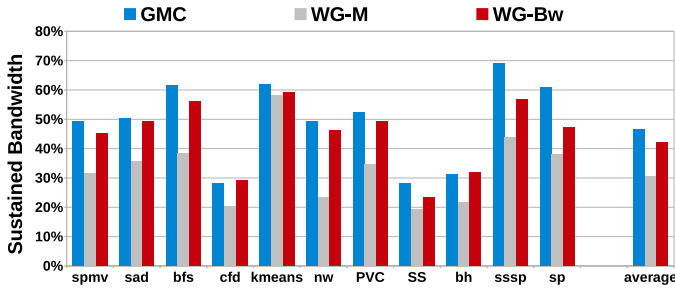Fig. 10. DRAM Latency Divergence with Different Schedulers

Fig. 12. Write Intensity

Fig. 11. Bandwidth Utilization of Different Schedulers

warps. The effective latency of a warp is the time taken for all of the warp's requests to be returned by the memory system. We see in Fig. 9, that the average effective latency is reduced by WG and WG-M by 9.1% and 16.9% respectively.

To further understand how the two different warp-aware scheduling schemes impact individual applications, we plot the average difference in the latency of the first and last request for a warp under the different scheduling schemes in Fig 10. This is a measure of the latency divergence for warps. With the two warp-aware scheduling schemes WG-M and WG, lower the latency divergence for a warp, lower is the memory stall time for that warp. The WG-M scheme is more effective than WG in reducing divergence for warps that spread their requests across multiple controllers. Applications *cfd, spmv, sssp, and sp* thus see higher performance with WG-M than WG as they touch 3.2 memory controllers on average. On the other hand, the WG scheme is effective in reducing the latency divergence for applications *sad, nw, SS*, and *bfs* that touch fewer than 2 controllers on average with every warp, and there is no further improvement for these applications with WG-M.

The benefit of the warp-aware scheduling scheme WG-M is further enhanced by incorporating the bandwidth optimization afforded by the MERB policy (WG-Bw). This is especially true for applications *bfs, PVC*, and *bh* where the reduction in bandwidth utilization with WG-M (see Fig. 11) negates some of the benefits of reduced latency divergence. The WG-Bw scheme improves bandwidth utilization of WG-M by more than 14% by overlapping row-misses with row-hits in other banks. Also, by servicing the row-miss from a warp shortly after its other requests are served, but not in strict succession, the WG-Bw scheme disrupts the latency divergence savings obtained by WG-M only marginally. In fact, the WG-Bw scheme achieves the best of both latency and bandwidth utilization and improves performance across all the benchmarks by 8.4% over the GMC.
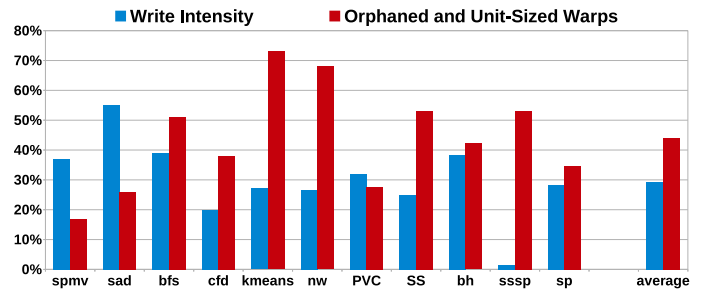
The WG-W mechanism ensures that long latency write-drains do not stall small warp-groups or partially served warp-groups and improves performance by 10.1% over the GMC. In Fig. 12, we plot the write-intensity i.e., the percentage of DRAM traffic that is composed of writes, and also the fraction of warp-groups stalled on a write-drain that are unit-sized, or contain orphaned requests. WG-W offers the largest improvements in applications where both these metrics are high e.g., *nw*, and *SS*. The warp-aware write-drain mechanism has no negative impact on the bandwidth utilization and only helps improve the average stall time for warps, thus improving WG-Bw further.

A. Impact on Non-divergent Applications

While the percentage of GPGPU applications with MAI are increasing, thanks to the deployment of the GPU in non-traditional arenas, it is important to ensure that legacy GPGPU applications, as well as applications that show little or no memory divergence are not impacted negatively by the new scheduling algorithms. We evaluated the impact of our proposals on a set of applications which are sensitive to DRAM performance, often being bandwidth-bound, and have structured data access patterns with significant intra-warp spatial locality. We looked at the following applications from Rodinia, MARS and Parboil suites for this purpose : *streamcluster, SRAD2 , Backprop-agation (bp), HotSpot , InvertedIndex*, and *PageViewRank*. These applications are sensitive to memory bandwidth, but have streaming memory access patterns that coalesce into one memory request in the common case. This is similar to the memory access patterns demonstrated by graphics applications as well. We see that with WG-W algorithm, there is a modest 1.8% performance improvement over the GMC-baseline for this set of benchmarks with no application suffering any slowdown. This result is not surprising. The scoring system for ranking warp-groups naturally favors streams of row-hit requests over row-misses and will thus work very similar to the GMC-baseline when there is only one request per warp (or few spatially colocated requests from each warp). This indicates why there is no degradation. The modest benefits are the result of the efficient row-miss scheduling under the WG-Bw policy and also from intelligent scheduling of write-drains.

B. Power and Energy Impact

The WG-W scheme has a 16% lower row-buffer hit rate compared to the GMC-baseline. We estimate the impact of this change on GDDR5 power consumption through a version

of the Micron power calculator [37] modified with current and voltage values from the simulated GDDR5 datasheet [23]. Since most of the power in GDDR5 chips is spent at the I/O drivers that drive the high-speed interface, the increased DRAM array access power due to increased row-misses increases the GDDR5 power by only 1.8%. In addition, compared to a GPU that consumes upwards of 200W of power, the memory controller consumes very little power. For example, a complex, programmable memory controller with 128KB of storage and multiple ALU-like pipelined paths [10] has been shown to consume only 152mW. Our proposals are significantly simpler than the PARDIS controller and hence will not have any noticeable effect on system power consumption. Once the improvement in system-throughput from our proposals is taken in account, there is a net improvement in energy consumption of the system.

C. Comparison with Other Memory Scheduling Mechanisms

1) Single-Bank Warp-Aware Scheduling (SBWAS): Lakshminarayana et al. proposed a SIMT-aware scheduling [32] for GPUs that incorporates information about the source of memory requests viz., warp id, and the available TLP in the issuing SM, to prioritize requests. The scheduler uses a potential function [46] to decide between issuing a row-hit request to a bank, and a row-miss request from a warp that has the fewest requests remaining. A parameter α controls the bias towards the latter of the two choices, and is determined empirically for different workloads.

Qualitatively, the four scheduling techniques and optimizations we propose have some fundamental differences with SBWAS. First, we prioritize warp-groups based on the *shortest completion time* and not the *shortest request count*. As discussed in Section III-B, this is important for irregular applications. Second, the MERB scheme can dynamically determine when to schedule a row-miss request from the selected warp so that it overlaps with row-hits in other banks. In SBWAS, the value of α is derived by profiling applications. Third, the algorithm in SBWAS applies only to a single bank. We manage the scheduling of requests of a warp across different banks of a channel (WG), and also across different memory channels (WG-M). Fourth, writes are interleaved with reads in SBWAS. We assume a more commonly used write-draining model which is higher performance and devise a mechanism to address warp-awareness during write-drain. Finally, the potential function in SBWAS requires a combination of complex calculations. The BASJF scheme and its derivatives require simple addition and comparison operations to select a warp-group.

We compare the WG-W scheme against against SBWAS for the workloads in Table V. For each benchmark, we determined the value of α by profiling (possible values being 0.25, 0.5, and 0.75). We found that on average, SBWAS provides an improvement of 2.51 % compared to the baseline GMC-baseline. The applications which generate requests to multiple banks and controllers (*e.g., spmv, sp, ssp, cfd*) show little improvement with SBWAS. *bfs* shows the most improvement with SBWAS as it touches fewer banks than other benchmarks (3.8 %). On the other hand, although the application *sad* generally touches one or two banks per warp, the high number

of writes erode the benefits of SBWAS when interleaved with reads.

2) Warp-Aware First-come First-served (WAFCFS): Yuan et al. [51] proposed complexity-effective memory scheduling where the interconnect between the SMs and the memory partitions does not interleave requests from different SMs. The expectation was that this would expose enough intra-warp spatial locality for the memory controller to use only a simple FCFS policy to harvest it. We model this policy using the WAFCFS scheduler which warp-groups in completion order. We observe that this leads to a 11.2% performance degradation compared to the GMC. While the WAFCFS scheduler can simplify scheduling for regular applications, in irregular applications, simple in-order servicing of requests from a warp can hardly achieve any row-hits.

3) CPU memory schedulers: Many ideas have been proposed for efficient memory scheduling in the CPU space. However, these techniques do not lend themselves to adoption for GPUs. For example, GPU workloads have lots of threads and warps with very similar memory demands. Thread Cluster Memory scheduling [31], that leverages the differences in memory intensities of threads, would not be as effective because distinguishing among warps is difficult based on memory demands alone.

PAR-BS: The PAR-BS scheme [40] forms batches of requests in a CPU's memory controller and issues requests from a batch to the memory system. The express motivation behind the batch-formation is fairness and as a result, a batch in PAR-BS will include requests from many threads and have different batches for different banks. Our batching scheme does exactly the opposite and groups requests from a warp together to reduce the latency divergence of a warp. In addition, we arbitrate between batches based on a bank-aware shortest job first policy to reduce wait time for warps while PAR-BS uses MLP to decide priorities.

ATLAS: The ATLAS scheduler [30] was proposed to promote fair-scheduling across channels in a multi-memory-controller CPU-chip. After a long time quanta, information is exchanged between the different memory controllers and a central arbiter to identify the threads that received the least service in the last quanta. These threads are prioritized in the next epoch. The ATLAS scheme uses long time quantas for scalability, whereas we need memory-controllers to co-ordinate at the granularity of warps. We avoid the complexities of sending requests to a central arbiter which has to maintain state for all controllers, by implementing a decentralized score update mechanism requiring little state information. In addition, none of the schemes described above mitigate the impact of the write-drain policy. Our warp-aware write-drain scheme provides significant benefits in many benchmarks.

VII. RELATED WORK

A. Memory Scheduling

A large body of work has looked at memory scheduling techniques for multi-core systems [14], [30], [31], [38], [40], [42]. Staged-Memory-Scheduling [6] aims to improve the bandwidth utilization of the DRAM channel in a heterogeneous CPU+GPU system by forming batches of row-hit requests

from each source and then arbitrating between these requests. Jeong et al. propose a QoS aware policy that allows the GPU to consume only the bandwidth that is absolutely necessary to maintain a certain QoS [24] and prioritize CPU requests to provide the latency sensitive CPUs with low latency. Recently, Jog et al. have proposed scheduling algorithms to better handle the fair allocation of the memory bandwidth between multiple concurrently executing kernels on different SMs [25]. However none of the proposed techniques have looked at the importance of incorporating warp-level ideas to reduce memory divergence. Prior work in the area of vector memory access mechanisms [17], [33] have shown that being cognizant of vector memory access patterns can lead to performance boosts. These proposals were motivated by erstwhile problems such as limited command and addressing bandwidth and their solutions (such as overlapping accesses from different vector loads in different banks, reordering requests from different vectors) are part of modern controllers that we improve upon.

B. Memory Divergence Mitigation in GPUs

Instead of utilizing warp-level multi-threading to hide the memory divergence latency, Meng et al. [35] advocate intra-warp latency hiding. This is accomplished through dynamic warp subdivision - a technique that allows some threads in a warp to make progress while the others are stalled on memory accesses [49]. This requires a single warp to be able to occupy multiple slots in the warp-scheduler and thus incurs at least double the cost and complexity in the scheduling hardware in each core. Several software optimizations have been proposed to tackle memory divergence. These include data herding [45] to force all threads in a warp to load from the same memory block through a compiler framework, a runtime system that tries to optimize the memory layout to reduce memory divergence [52] as well as techniques to improve memory coalescing [16]. Recently other techniques have been proposed to reduce effective memory latency [26], [27], [43].

VIII. CONCLUSIONS

To leverage the energy-efficiency of SIMT parallel-processors for executing parallel applications from diverse domains, it is important to reduce the effects of memory latency divergence. In this paper, we quantified the impact of DRAM latency divergence on irregular GPGPU workloads and proposed scheduling schemes that can maintain bandwidth while reducing divergence to boost performance by 10.1%. This paves the way for other memory scheduling techniques that are cognizant of the intricacies of the SM cores - for example, prioritizing warp-groups that contain blocks of data that are shared by multiple warps.

REFERENCES

[1] "NVIDIA Kepler GK110 Whitepaper," 2012, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[2] "Intel Architecture Instruction Set Extensions Programming Reference," 2013, http://download-software.intel.com/sites/default/files/319433-016.pdf.

[3] T. M. Aamodt and W. L. Fung, "GPGPU-Sim 3.x Manual," http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual.

[4] ——, "GPGPU-Sim Accuracy," http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual#Accuracy.

[5] "The Opportunities and Challenges of Exascale Computing," http://science.energy.gov/~/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf, 2010.

[6] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Hetergenous Systems," in *Proceedings of ISCA*, 2012.

[7] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W. W. Hwu, "Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors," in *Proceedings of PPoPP*, 2012.

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

[9] E. Blem, M. Sinclair, and K. Sankaralingam, "Challenge Benchmarks That Must be Conquered to Sustain the GPU Revolution ," in *Proceedings of EAMA-4*, 2011.

[10] M. Bojnordi and E. Ipek, "PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards," in *Proceedings of ISCA*, 2012.

[11] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of IISWC*, 2012.

[12] M. Burtscher and K. Pingali, "An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-body Algorithm," in *GPU Computing Gems Emerald Edition, Morgan Kaufmann*, 2011.

[13] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, UUCS-12-002.

[14] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi, "Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads," in *Proceedings of HPCA*, 2012.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of IISWC*, 2009.

[16] S. Che, J. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," in *Proceedings of SC*, 2011.

[17] J. Corbal, R. Espasa, and M. Valero, "Command Vector Memory Systems: High Performance at Low Cost," in *Proceedings of PACT*, 1998.

[18] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke, "PEPSC: A Power-Efficient Processor for Scientific Computing," in *Proceedings of PACT*, 2011.

[19] "Green 500 List - Nov 2013," http://www.green500.org/lists/green201311, Nov 2013.

[20] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *Proceedings of PACT*, 2008.

[21] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems," in *Proceedings of ISPASS*, 2012.

[22] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *Proceedings of ISCA*, 2009.

[23] "Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0 ," http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf, Hynix, 2009.

[24] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *Proceedings of DAC*, 2012.

[25] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. Kandemir, and C. R. Das, "Application-aware Memory System for Fair and Efficient Execution for Concurrent GPGPU Applications," in *Proceedings of GPGPU-7*, 2014.

[26] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. Das, "Orchestrated Scheduling and Prefetching for GPUs," in *Proceedings of ISCA*, 2013.

[27] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. Das, "OWL: Cooperative Thread

Array Scheduling Techniques for Improving GPGPU Performance," in *Proceedings of ASPLOS*, 2013.

[28] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, 2011.

[29] Khronos Group, "OpenCL," http://www.khronos.org/opencl.

[30] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *Proceedings of HPCA*, 2010.

[31] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *Proceedings of MICRO*, 2010.

[32] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin, "DRAM Scheduling Policy for a GPGPU Architecture Based on a Potential Function," in *IEEE Computer Architecture Letters*, Nov 2011.

[33] B. Matthew, S. A. McKee, J. B. Carter, and A. Davis, "Design of a Parallel Vector Access Unit for SDRAM Memory Systems," in *Proceedings of HPCA*, 2000.

[34] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU Implementation of Inclusion-based Points-to Analysis," in *Proceedings of PPoPP*, 2012.

[35] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch amd Memory Divergence Tolerance," in *Proceedings of ISCA*, 2010.

[36] D. G. Merrill, M. Garland, and A. S. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of PPoPP*, 2012.

[37] *Calculating Memory System Power for DDR3 - Technical Note TN-41-01*, Micron Technology Inc., 2007.

[38] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *Proceedings of MICRO*, 2007.

[39] NVIDIA Corporation, "NVIDIA Cuda C Programming Guide v4.2," http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[40] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling - Enhancing Both Performance and Fairness of Shared DRAM Systems," in *Proceedings of ISCA*, 2008.

[41] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," in *Proceedings of ASPLOS*, 2014.

[42] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory Access Scheduling," in *Proceedings of ISCA*, 2000.

[43] T. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious Wavefront Scheduling," in *Proceedings of MICRO*, 2012.

[44] ——, "Divergence-aware Warp Scheduling," in *Proceedings of MICRO*, 2013.

[45] J. Sartori and R. Kumar, "Branch and Data Herding: Reducing Control and Memory Divergence for Error-tolerant GPU Applications," in *IEEE Transactions on Multimedia*, 2012.

[46] D. Shah and D. Wischik, "Switched networks with maximum weight policies: Fluid Approximation and Multiplicative State Space Collapse," *The Annals of Applied Probability*, vol. 22, 2012.

[47] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W.-. M. W. Hwu, "The Parboil Technical Report," University of Illinois, Tech. Rep., 2012.

[48] J. Stuecheli, D. Kaseridis, D. Daly, H. Hunter, and L. John, "The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies," in *Proceedings of ISCA*, 2010.

[49] D. Tarjan, J. Meng, and K. Skadron, "Increasing Memory Miss Tolerance for SIMD Cores," in *Proceedings of SC*, 2009.

[50] Z. Wang, S. M. Khan, and D. A. Jimenez, "Improving Write-Back Efficiency with Decoupled Last-Write Prediction," in *Proceedings of ISCA*, 2012.

[51] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures," in *Proceedings of MICRO*, 2008.

[52] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly Elimination of Dynamic Irregularities for GPU Computing," in *Proceedings of ASPLOS*, 2012.

[53] Z. Zhang, Z. Zhu, and X. Zhand, "A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality," in *Proceedings of MICRO*, 2000.