Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications

Adwait Jog[1*] Evgeny Bolotin[2] Zvika Guz[2] Mike Parker[3*]
Stephen W. Keckler[2,4] Mahmut T. Kandemir[1] Chita R. Das[1]
[1]The Pennsylvania State University [2]NVIDIA [3]Intel Corp. [4]The University of Texas at Austin
(adwait, kandemir, das)@cse.psu.edu, (ebolotin, zguz, skeckler)@nvidia.com, mike.a.parker@intel.com

ABSTRACT

The available computing resources in modern GPUs are growing with each new generation. However, as many general purpose applications with limited thread-scalability are tuned to take advantage of GPUs, available compute resources might not be optimally utilized. To address this, modern GPUs will need to execute multiple kernels simultaneously. As current generations of GPUs (e.g., NVIDIA Kepler, AMD Radeon) already enable concurrent execution of kernels from the same application, in this paper we address the next logical step: executing multiple concurrent applications in GPUs. We show that while this paradigm has a potential to improve the overall system performance, negative interactions among concurrently executing applications in the memory system can severely hamper the performance and fairness among applications. We show that the current application agnostic GPU memory system design can (1) lead to sub-optimal GPU performance; and (2) create significant imbalance in performance slowdowns across kernels. Thus, we argue that GPU memory system should be augmented with application awareness. As one example to the applicability of this concept, we augment the memory system hardware with application awareness such that requests from different applications can be scheduled in a round robin (RR) fashion while still preserving the benefits of the current first-ready FCFS (FR-FCFS) memory scheduling policy. Evaluations with different multi-application workloads demonstrate that the proposed memory scheduling policy, first-ready round-robin FCFS (FR-RR-FCFS), improves fairness and delivers better system performance compared to the existing FR-FCFS memory scheduling scheme.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

*The work was done while working at NVIDIA Research, Santa Clara, CA, during Summer 2013.

GPGPU-7, March 01 2014, Salt Lake City, UT, USA

http://dl.acm.org/citation.cfm?doid=2576779.2576780.

General Terms

Design, Performance

Keywords

GPGPUs; Memory System; CUDA Streams

1. INTRODUCTION

Graphics Processing Units (GPUs) are expected to play a critical role in the foreseeable computing landscape ranging from supercomputing machines to hand-held devices. These GPUs are characterized by numerous programmable computational cores and thousands of simultaneously active fine-grained threads. Traditionally, GPUs were designed to execute only a *single* kernel at a time; it was expected that a single kernel would have enough threads to keep all GPU resources busy. However, as GPU resources are growing with each new generation (for example, the state-of-the-art Kepler Architecture model GTX 780 Ti has 2880 cores [3], and AMD Radeon R9 290X has 2816 cores [1]), and as more irregular and general-purpose applications are ported to GPUs, many kernels will not be able to proportionally scale [19] and effectively utilize the growing compute resources. To address this problem, a new GPU computing paradigm was recently introduced, where multiple kernels can be executed concurrently on the same GPU platform. This paradigm has two primary advantages. First, it significantly improves the GPU efficiency, as shown in Fermi White Paper [18] and Pai et al. [19]. Second, it facilitates the consolidation of jobs from multiple independent users on to the same GPU substrate, as demonstrated by NVIDIA GRID technology [2].

Figure 1 shows this new computing paradigm pictorially. Figure 1 (A) shows the traditional GPU architecture, where all cores are executing a single kernel. In Figure 1 (B), the same GPU architecture is concurrently executing multiple kernels. Broadly speaking, these multiple kernels can originate from: (i) a single application, or/and (ii) multiple independent applications (contexts). Although this new computing paradigm is an effective way to increase GPU performance and resource utilization, many architectural challenges need to be addressed to unlock its full potential. Open issues that have not been sufficiently explored include: efficient hardware support for execution of multiple applications[1], finding the optimal number of cores for a particular kernel, designing a QoS-aware on-chip network

[1]In this paper, we assume concurrently executing kernels originate from separate applications. Hence, we use the terms kernels and applications interchangeably.

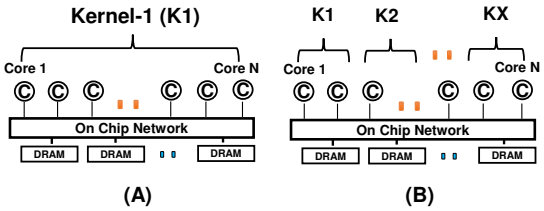

Figure 1: Baseline GPU architecture executing: (A) Single kernel, (B) Multiple kernels concurrently.

fabric and memory hierarchy, augmenting the memory hierarchy to include resource sharing policies, and concurrency management for effectively utilizing the on-chip shared resources. Given the wide scope of the problem, this paper focuses on one aspect of the design space: management and allocation of shared memory system resources in GPUs executing multiple applications concurrently.

Our goal is to provide a better understanding of the interactions of concurrent applications in the GPU memory subsystem. We observe that the traditional GPU memory system is application agnostic, and the widely used First-ready FCFS (FR-FCFS) [20,21,25] memory scheduler is unaware of the individual demands of concurrent multiple applications. It primarily focuses on improving DRAM page hit rates, and might hurt overall system performance and fairness. This problem becomes more pronounced when the memory demands of concurrently executing applications have wide variation, implying that an application with high memory demand attempts to monopolize the resource usage over an application with low memory demands. To address this, we augment application-awareness to the memory-system logic and propose the First-ready Round-robin FCFS (FR-RR-FCFS) memory scheduler, which schedules memory requests originating from different applications in a round-robin (RR) fashion, while preserving the benefits of FR-FCFS scheduling. We show that this simple change to the existing FR-FCFS memory scheduler is a better alternative for concurrent execution of multiple kernels in terms of equitable memory bandwidth sharing and overall system performance.

This paper makes the following **contributions**:

- To the best of our knowledge, this is the first work that provides a detailed analysis of the interactions of multiple applications in the GPU memory system.
- This work manifests the fact that a naive coupling and execution of different applications concurrently on modern GPUs with application-agnostic shared resource allocation do not lead to desired results.
- We show that one of the primary reasons for sub-optimal performance and unfairness is the application agnostic management of shared resources. The popular FR-FCFS memory scheduler fails to distinguish memory requests originating from multiple applications.
- In this context, we propose to propagate application awareness to the memory system scheduling logic. As one possible implementation, we suggest a memory controller scheduling policy which not only preserves the benefits of FR-FCFS, but also improves fairness and overall system performance by serving memory requests of different applications in an round-robin fashion. We evaluate the proposed memory scheduler across 14 diverse 2-application workloads on a 60-core GPU simulated platform using GPGPU-Sim [6]. On average, we observe 7% improvement (up to 49%) in fairness, 10% improvement (up to 64%) improvement in instruction throughput performance, and up to 7% improvement in weighted system speedup.

2. BACKGROUND AND EXPERIMENTAL METHODOLOGY

In this section, we provide a brief description of the baseline GPU architecture and experimental methodology.

2.1 Baseline GPU Architecture

Our baseline GPU (see Figure 1) consists of multiple cores, named Streaming Multiprocessors (SMs). Each SM is associated with a private L1 data cache, a read-only texture cache and a constant cache. A software managed scratchpad memory is also associated with each SM. SMs are connected to memory channels (partitions) via a crossbar, and memory requests to each partition are handled by a GDDR5 memory controller. We simulate the baseline architecture described in Table 1 using GPGPU-Sim v3.2.1 [6], a cycle-accurate GPU simulator.

Single Application Scheduling: A typical CUDA application consists of multiple kernels (or grids). Each kernel is further divided into groups of threads, called cooperative thread arrays (CTAs). Traditionally, GPUs execute all kernels of an application sequentially, i.e. one kernel at a time. In this scenario, when a kernel is launched on the GPU, the CTA scheduler picks available CTAs associated with that kernel and distributes them to the SMs as evenly as possible [6]. The maximum number of CTAs per SM is limited by various resources associated with each SM, and by the resources required by a given kernel [6, 15]. Hence, if a kernel requires less resources, the maximum number of CTAs per SM will be larger than that of another kernel whose CTAs need more resources.

Multiple Application Scheduling: In this paper we consider the case where multiple *kernels* from multiple *applications* are executed concurrently, i.e., we simultaneously execute kernels from *different* applications. Since the focus of the paper is the memory system, we use a simple kernel-to-SM assignment scheme: in a two-application scenario where two kernels of different applications are executed concurrently, we assign half of the SMs to the first application, and the second half to the other application. We leave sophisticated SM-partitioning techniques as future work. The CTA assignment for each kernel follows the same load-balanced distribution strategy as described before; the only difference is that each kernel is now assigned to only half of the SMs of the baseline GPU architecture.

Memory Scheduling in GPUs: First-ready FCFS (FR-FCFS) [20, 21, 25] is the commonly employed memory scheduling technique in GPUs. This scheme is targeted at improving DRAM row hit rates, so request prioritization order is as follows: 1) row-hit requests are prioritized over other requests; then 2) older requests are prioritized over younger requests. Among row-hit requests, older requests are prioritized over younger requests.

2.2 Evaluation Methodology

The new generation of GPUs allows concurrent execution of *streams*, where a *stream* is defined as a set of commands

Table 1: Simulated baseline GPU configuration

SM Configuration	1400MHz, SIMT width = 16×2
Resources / SM	Max. 1536 threads (48 warps, 32 threads/warp), 48KB shared memory, 32684 registers
Caches / SM	16KB 4-way L1 data cache, 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
Default Warp Scheduling	Greedy-then-oldest (GTO) [22]
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Interconnect	1 crossbar/direction (60 SMs, 6 Memory Controllers), 1400MHz
Memory Model	6 GDDR5 Memory Controllers (MCs), First-Ready FCFS (FR-FCFS) scheduling 16 DRAM-banks/MC, 924 MHz memory clock
Hynix GDDR5 Timing [10]	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$

required to be executed serially. We exploit this mechanism to issue commands from different applications (kernels) to separate streams, thus allowing them to execute concurrently. We study 6 CUDA applications; Table 2 lists the applications along with their DRAM bandwidth utilization. Note that the considered applications have diverse memory demands – while `GUPS` has the highest memory bandwidth utilization of 93%, `HIST` and `DGEMM` have considerably lower memory bandwidth utilization (50% and 34%, respectively). In Section 3, we show that memory intensive applications like `GUPS` significantly interfere with co-scheduled applications, leading to poor overall performance and fairness.

From these 6 CUDA applications, we form all possible two-application workloads and simulate them on the GPGPU-Sim simulator. We omit the results of one workload (`bfs_dgemm`) as our infrastructure could not simulate it faithfully. Table 3 lists all 14 two-application workloads. We collect statistics at the point when both applications execute to completion at least once. To do so, if one of the applications finishes execution earlier than the other, we relaunch the faster running application again. This process continues until the slower running application completes.

Table 2: Evaluated applications, along with their DRAM bandwidth utilization when they are executed alone on the entire baseline GPU architecture.

Application	Abbr.	Bandwidth Utilization
Histogram	HIST	50%
Gaussian	GAUSS	70%
Random Access	GUPS	93%
Breadth First Search	BFS	79%
3D Stencil	3DS	85%
Matrix Multiplication	DGEMM	34%

2.3 Evaluation Metrics

We report on: (I) Weighted Speedup, for measuring application throughput, (II) Instruction Throughput, for measuring raw machine throughput, and (III) Fairness Index, for measuring fairness in the system. For weighted speedup (WS), we measure the slowdown experienced by each application relative to the case where it runs alone on the entire GPU (Eq.(1)). Note that when an application is running alone, it can use all SMs in the system. The sum of slowdowns of all the concurrent applications is defined as weighted speedup (Eq.(2)). WS indicates how many jobs are executed per unit time. Assuming there is no constructive interference among applications, the maximum value of WS is equal to the number of applications. Thus, in a 2-application mix, the optimal (maximum) value of WS is 2. In the worst case, if both the applications stall the progress of each other indefinitely, WS will be equal to 0. Instruction Throughput (IT) is defined as the total number of instructions committed per cycle in the entire chip. (Eq.(3)). It basically depicts the raw machine throughput. We use Fairness Index (FI) to express the imbalance of performance slowdowns across applications. Eq.(4) shows the FI equation for a system that executes two application concurrently. When FI is equal to 1 the system is completely fair, because all applications are slowed down equally when they execute concurrently and share the same resources.

Summary of Evaluation Metrics

(A) IPC_i is the number of committed instructions per cycle (IPC) of i^{th} application, **(B)** IPC_i^{shared} is IPC of i^{th} application when it is co-scheduled with other applications, **(C)** IPC_i^{alone} is IPC of i^{th} app. when it is the only application executing on the entire GPU,

Eq.(1) $Slowdown(APP_i) = \frac{IPC_i^{shared}}{IPC_i^{alone}}$

Eq.(2) $Weighted\ Speedup = \sum_i Slowdown(APP_i)$

Eq.(3) $Instruction\ Throughput = \sum_i IPC_i$

Eq.(4) $Fairness\ Index = MAX(\frac{slowdown(APP_1)}{slowdown(APP_2)}, \frac{slowdown(APP_2)}{slowdown(APP_1)})$

Table 3: Evaluated 2-application GPU workloads.

Workload #	1st APP	2nd APP	Abbr.
1	HIST	GAUSS	hist_gauss
2	HIST	GUPS	hist_gups
3	HIST	BFS	hist_bfs
4	HIST	BFS	hist_bfs
5	HIST	3DS	hist_dgemm
6	GAUSS	GUPS	hist_gups
7	GAUSS	BFS	gauss_bfs
8	GAUSS	3DS	gauss_3ds
9	GAUSS	DGEMM	gauss_dgemm
10	GUPS	BFS	gups_bfs
11	GUPS	3DS	gups_3ds
12	GUPS	DGEMM	gups_dgemm
13	BFS	3DS	bfs_3ds
14	3DS	DGEMM	3ds_dgemm

3. CONCURRENT KERNEL EXECUTION CHALLENGES

In this section, we zoom in on the memory system and highlight the main challenges associated with concurrent execution of applications. We show that while concurrent execution of applications can increase overall system performance, negative interactions among applications in the memory system can hamper application performance, and can introduce severe fairness problems.

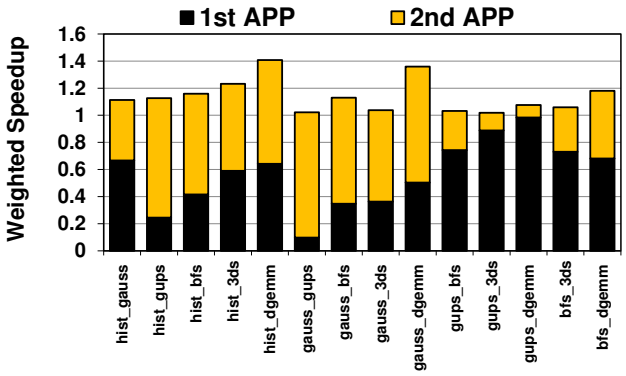

Figure 2: Weighted speedup (Application throughput) for the evaluated workloads. The `1st APP` and `2nd APP` are the first and second applications in the workload, respectively, as mentioned in Table 3.

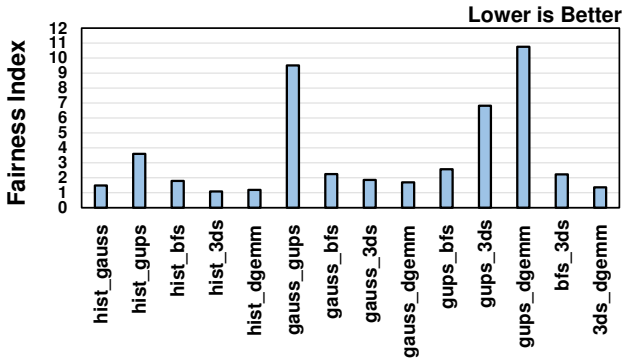

Figure 3: Fairness Index for the evaluated workloads when the memory scheduler adopts the baseline FR-FCFS scheduling policy.

3.1 Fairness considerations

Explicitly addressing fairness in the memory system is essential whenever multiple applications (potentially from different users) share the same resources. An unfair or uncoordinated resource allocation can lead to imbalance in performance degradation across applications. This phenomenon is demonstrated in Figure 2, where we plot the weighted speedups obtained for all 14 evaluated workloads. We show the slowdown of each application in every workload. `1st APP` shows the slowdown of the first application when co-scheduled with the `2nd APP`, and vice versa. In a completely fair system, the values of slowdowns (performance degradation) for each one of the two applications would be the same. Instead, we observe that this is hardly ever the case, and the slowdowns are considerably different between the two applications. For example, consider the case of `gups_dgemm`, where there is significant difference between the slowdowns of `GUPS, 1st APP`, and `DGEMM, 2nd APP`. This means that while GUPS performance is hardly affected by sharing the GPU resource, DGEMM performance is considerably degraded. In fact, most of the contribution to WS in this case is associated with GUPS.

Figure 3 shows the FI metric for the evaluated workloads. While slowdowns in some workloads are relatively balanced (e.g., `hist_dgemm`) the FI value for other workloads is very high (e.g. `gups_3ds` and `gauss_gups`). Particularly, the FI for `gups_dgemm` is the worst (maximal) among all the workloads (10.75). The reason for such high FI value is the fact that `GUPS` has very high memory bandwidth demands compared to other co-scheduled applications. Its memory bandwidth utilization reaches 93% (see Table 2), which significantly degrades performance of other co-scheduled applications. As we will show next, the presence of such memory bandwidth demanding applications causes imbalance in performance degradation, leading to very high FI values.

To get a deeper insight into the mechanics of the cross-application interference in the memory system, Figure 4 shows a break down of the memory bandwidth to the following components: (A) 1st and 2nd APP: the relative portion of DRAM cycles during which the 1st and 2nd applications in the workload move useful data over the DRAM interface, (B) Wasted-BW: the relative portion of DRAM cycles during which no data is transferred over the DRAM interface, but there are pending memory requests in memory controller. This is because of DRAM timing constraints; improving DRAM page hit rates, and bank-level parallelism can reduce this Wasted-BW, and (C) Idle-BW: the relative portion of DRAM cycles during which there are no requests pending in the memory controller queues, and hence DRAM is idle. Besides the concurrent execution configuration, the figure also plots alone_60 – where an application executes in a stand-alone mode over the entire 60 SM GPU system, and alone_30 – where an application executes in a stand-alone mode over half of the compute resources (up to 30 SMs in our case).

It is evident from Figure 4 that the memory intensive applications monopolize the memory scheduler while the lighter applications are unable to get a fair share of the bandwidth. For example, when `GUPS` is co-scheduled with other applications (`HIST, GAUSS, BFS, 3DS, and DGEMM`), the majority of the bandwidth is consumed by `GUPS`, while the other application gets a very small share of the bandwidth. In the case of `gups_dgemm`, the memory bandwidth obtained by `GUPS` reduces only marginally (by 6% over alone_60 configuration), but `dgemm` achieves only 3% of the memory bandwidth (31% lower than its alone_60 configuration). This imbalanced allocation of memory resources translates to imbalance in performance degradation – GUPS slows downs by only 2% while `DGEMM` slows down by 90%, when `GUPS` and `DGEMM` are coupled together. Overall, these observations indicate that one of the main reasons for poor fairness is the interference caused by applications with intensive bandwidth requirements.

3.2 Throughput considerations

One of the primary motivations for preferring concurrent execution of multiple applications over time division multiplexing of the GPU hardware is to increase the machine utilization and thus improve application throughput. Applications throughput is reflected by weighted speedup and is shown in Figure 2. Indeed, the achieved WS for each one of the workloads is above one, indicating that concurrent execution performs either as good or better than a time division scheme. Some workloads present significant speedups - up to 41% for `hist_dgemm`. This is because these two applications have the lowest memory bandwidth demands in our workload suite (see Table 2), and hence do not interfere significantly in the memory system when co-scheduled

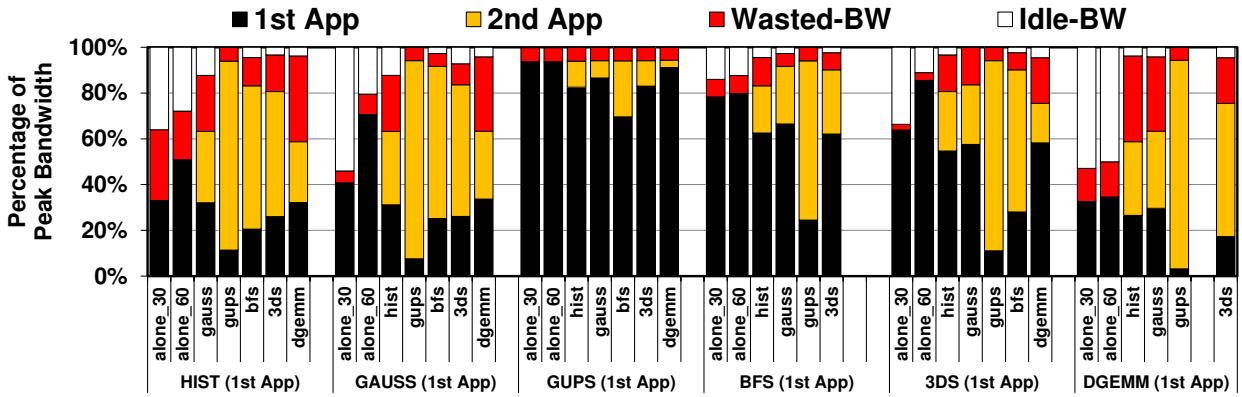

Figure 4: DRAM bandwidth utilization distribution across various workloads when memory scheduler adopts the baseline FR-FCFS memory scheduling policy.

together. However, other workloads present modest to minimal gains: in the case of `gauss_gups`, the interference is significant and is mostly caused by `GUPS`, leading to only 2% improvement in WS. In fact, most applications that are co-scheduled with `GUPS` exhibit poor WS.

The main reasons for this sub-optimal weighted-speedup behavior is in fact a manifestation of the previously discussed fairness problem, where the bandwidth-intensive application significantly degrades the performance of other applications leading to lower overall weighted speedup. For example, consider the case of `gups_dgemm` where the weighted-speedup is only 7%. `GUPS` interferes with the progress of `GAUSS` leading to its sub-optimal performance resulting in lower overall WS.

4. APPLICATION-AWARE MEMORY SCHEDULING

From the analysis in Section 3, we observe that application-agnostic approach of the underlying memory-system scheduling policy leads to sub-optimal results both in terms of overall application throughput and fairness. We believe that it is imperative to develop an application-aware memory scheduling approach to address these issues. To this end, in this section we propose and discuss details of an example design of a simple application-aware memory scheduler that improves fairness and performance. We also discuss its associated hardware overheads.

4.1 Designing Application-aware Memory Scheduler

In previous section we discussed the negative interference phenomena among bandwidth-intensive and the light applications on a single GPU. The bandwidth-intensive applications (e.g. `GUPS`) can severely degrade the performance of its co-scheduled applications, leading to sub-optimal application performance and fairness. To address these issues, we propose an example implementation of a simple memory scheduler. We propose to equip the widely known FR-FCFS memory scheduler with application awareness by choosing requests from different applications in round-robin (RR) fashion. The advantage of this memory scheduling approach is that the bandwidth-intensive application would not be able to starve its co-scheduled applications for a long period time. Note that our scheme still prioritizes the row-hit requests over other requests for optimizing DRAM page hit rates. The only difference in our memory scheduler is that the requests are not picked in FCFS fashion, but in RR fashion across applications. If the pending memory controller queue has only requests from either one of the applications, the RR automatically performs FCFS.

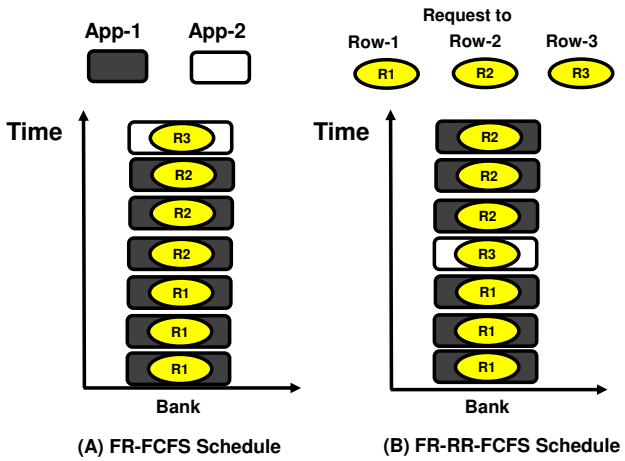

Figure 5: Conceptual example showing the working of (A) baseline FR-FCFS memory scheduling, (B) our proposed FR-RR-FCFS memory scheduling.

Formally, we propose First-ready Round-robin FCFS (FR-RR-FCFS) memory scheduling method for handling memory requests from multiple GPU applications. The request prioritization order of FR-RR-FCFS is: 1) row-buffer-hit requests over all other requests, 2) requests from the application next in the round-robin scheduling order, 3) older requests over younger ones. Among row-hit requests, older requests are prioritized over younger requests.

Figure 5 shows the mechanics of the proposed FR-RR-FCFS memory scheduling function, for multiple concurrent GPU applications. Figure 5 (A) shows the baseline scheduler with FR-FCFS memory scheduling function. Without the loss of generality, in this example, we assume one memory bank and one-memory controller memory system. Furthermore, we assume that two applications, App-1 and App-2, are concurrently executing on the same GPU platform. The

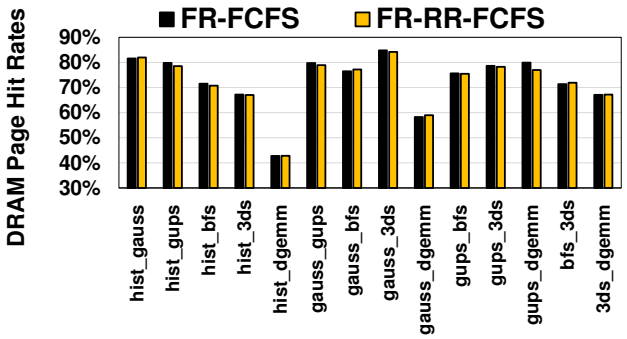

Figure 6: Effect on DRAM page hit rates. The proposed scheduler FR-RR-FCFS preserves the DRAM page hit rates obtained by the baseline FR-FCFS memory scheduler.

memory requests are tagged with their application-id (in this example, they are color coded – App-1 (gray) and App-2 (white)). In Figure 5 (A), the first memory request arriving to the memory controller is originating from App-1, and destined to row-1 (R1). Similarly, the next two requests also belong to R1, and originate from App-1. Because of the FR-FCFS policy, the baseline memory scheduler will schedule all these three requests back-to-back. The next three requests are also originated from App-1 (also gray coded), but are destined to R2. As the memory scheduler is application-agnostic, it will keep on scheduling those requests in their arrival order, and the request from App-2 (the last request – white coded, destined to R3) would be delayed significantly in that case. Alternatively, FR-RR-FCFS memory scheduler in Figure 5 (B), would service App-2 request after servicing first three App-1 requests destined to R1. In our proposed policy the waiting time of memory requests from App-2 is substantially shorter, and thus it will not be starved by requests from App-1.

One might argue that after servicing the first memory request of App-1, memory scheduler should shift to App-2 for the natural round-robin sequence. However, by doing so, the scheduler would have to switch the memory-row (from R1 to R3) and back to R1 (R3 to R1). These row-switches would have degraded DRAM page hit-rates and throughput. In order to preserve DRAM page-hit rate, our scheme first services all the memory requests to the same page, and then moves to the next application in round-robin fashion. Figure 6 shows that the DRAM page hit-rates for FR-FCFS and our scheduler are roughly the same (average reduction is less than 1%).

4.2 Hardware Complexity

The proposed method is relatively simple to implement in hardware, and that it would require a very low additional hardware cost compared to an existing scheduling logic.

In order to propagate application-related information throughout the memory-system, the memory request need to be tagged with the application-id information. The tagging is performed at the SM-level. For a limited number of concurrent applications on a single GPU, we assume several bits per memory request. For the example discussed in this paper, of up to 2 applications per GPU, single bit is needed for application-id extension of the request meta-data fields.

The additional hardware required for the RR function in the memory controller is minimal compared to an existing FR-FCFS logic. It requires a duplication of the find-first masking logic according to the application ID, similar to what is done for finding the first ready request for an open-row in the memory controller already. In addition, it is required to compute the next-application ID in a RR fashion, which can be implemented by a simple rotating function. Note that all the required information is computed locally at the memory controller, and no communication/coordination across memory controllers, and banks within memory controller is required.

5. EXPERIMENTAL RESULTS

In this section, we provide comparative analysis of the evaluated schedulers in terms of fairness and performance.

5.1 Fairness Results

Figure 7 shows the fairness index (FI) for all the evaluated workloads, both for the baseline memory scheduler (FR-FCFS), and for our proposed FR-RR-FCFS policy. We observe significant improvements in fairness (decrease in FI) with FR-RR-FCFS: 49% for `hist_gups`, 47% for `gups_3ds`, 14% for `gauss_bfs`, and 11% for `hist_bfs`. On average, we observe 7% improvement in fairness over the baseline FR-FCFS policy. To understand these benefits better, we re-plotted Figure 4 in Figure 8, but here we compare the bandwidth distribution of the baseline FR-FCFS with the distribution achieved when using proposed FR-RR-FCFS scheduling policy. For clarity, we have omitted the workloads that do not have significant difference in these distributions. We observe that improvement in FI has originated from a fairer distribution of the overall memory bandwidth across the concurrently executing kernels. FR-RR-FCFS provides more memory bandwidth to the *lighter* applications (compared to the baseline), and thus limits their performance degradation. For example, when `HIST` is coupled with `GUPS`, the bandwidth obtained by `HIST` is increased from 10% (fr-fcfs-gups) to 20% (fr-rr-fcfs-gups). Note that, ideally `HIST` should reach up to 33% and 50%, when it executes alone on 30 and 60 SMs system, respectively. Our proposed scheduler has facilitated in bridging this gap.

5.2 Performance Results

Figure 9 and Figure 10 show the improvement in instruction throughput (IT) and weighted-speedup (WS) when using the FR-RR-FCFS memory scheduler, respectively. Results are normalized to the baseline FR-FCFS scheduler. We observe significant improvement in IT and WS for workloads `hist_gups`, `hist_bfs`, `gauss_gups`, and `gups_3ds`. The maximum improvement is observed in `hist_gups`: 64% in instruction throughput and 7% in weighted speedup. These improvements in performance are primarily due to the fairer allocation of memory bandwidth (see Figure 8). FR-RR-FCFS facilitates the lighter applications and thus reduces their performance degradation. It is evident that the high memory demanding applications `GUPS` and `BFS` are now comparatively less dominant, thereby improving performance.

We note that in two cases (`hist_gauss` and `hist_3ds`), there is a small decrease (2-3%) in WS when using FR-RR-FCFS. Clearly, in these workloads, the FR-RR-FCFS scheduler is unable to intelligently allocate memory bandwidth among the applications. Indeed, while "round-robin"

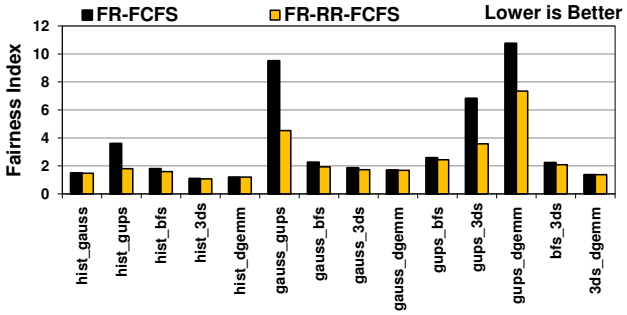

Figure 7: Fairness index (FI) of the evaluated workloads when memory scheduler adopts FR-FCFS (baseline, 1st bar) and FR-RR-FCFS (proposed, 2nd bar) scheduling techniques.

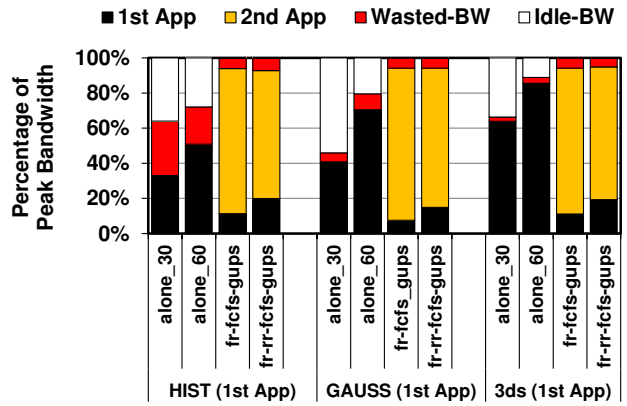

Figure 8: DRAM bandwidth utilization distribution across selected workloads when memory scheduler adopts FR-FCFS (baseline, 3rd bar) and FR-RR-FCFS (proposed, 4th bar) scheduling techniques.

gives better opportunity to all concurrent applications in taking service from memory, it is still unaware of individual application's *characteristics*. A more sophisticated scheme might use application characterization to influence the priority settings in attaining service from memory. We leave the design of such schemes for future works.

6. RELATED WORK

To the best of our knowledge, this is the first work to provide a detailed analysis on the interactions of multiple applications in GPU memory system, and propose a memory scheduler to improve both fairness and overall performance.

Memory scheduling techniques: There is a large body of work on memory scheduling techniques in the context of multi-cores. Thread cluster memory scheduling (TCM) [14] classified applications on the basis of their sensitivity to memory bandwidth and latency. They further proposed various memory request prioritization schemes for improving fairness and throughput. However, their work only considers multiple single-programmed applications. On the other hand, our work focuses on multiple massively threaded applications, and proposes a simple but effective memory scheduling technique to handle their memory requests. Ebrahimi et al. [7] proposed parallel application

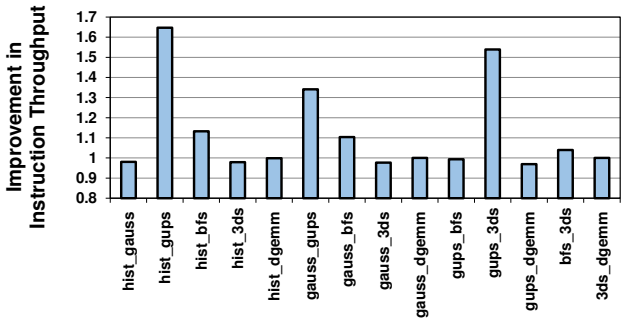

Figure 9: Improvement in instruction throughput (IT) across the evaluated workloads. Results are normalized to the case when memory scheduler adopts the baseline FR-FCFS scheduling policy.

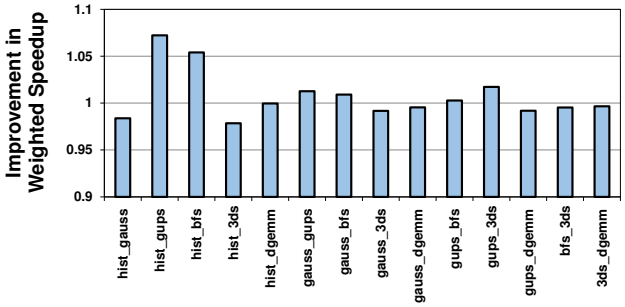

Figure 10: Improvement in weighted speedup (WS) across the evaluated workloads. Results are normalized to the case when memory scheduler adopts the baseline FR-FCFS scheduling policy.

memory scheduling, where they explicitly managed inter-thread memory interference for improving performance, especially in critical sections of the program. However, their work only considers a single multi-threaded application, while our work deals with scheduling of multiple multi-threaded applications.

In the context of GPUs, Lakshminarayana et al. [16] explored a DRAM scheduling policy that essentially chooses between Shortest Job First (SJF) and FR-FCFS [21, 25]. Yuan et al. [24] proposed an arbitration mechanism in the interconnection network to restore the lost row buffer locality caused by the interleaving of requests. They showed that performance of in-order DRAM memory scheduler can be competitive to FR-FCFS, if interconnect is aware of the requests destined to the same row. Ausavarungnirun et al. [5] proposed a staged memory scheduler for CPU-GPU architectures. Their primary goal was to improve row-buffer locality in heterogeneous architectures. All these works only focus on improving the performance of single GPU application and do not focus on the scenarios when multiple applications are scheduled concurrently, as we do in our work.

Concurrent execution of multiple kernels on GPUs: Pai et al. [19] proposed elastic kernels that allow a fine-grained control over their resource usage. Further, they proposed elastic-kernel aware concurrency management policies for improving GPU performance. Adriaens et al. [4] proposed spatial partitioning of SM resources across concurrent applications. They presented a variety of heuristics for di-

viding the SM resources across applications. In our work, we assume an even partitioning technique, according to which SMs are distributed evenly among concurrent applications. Gregg et al. [9] presented KernelMerge, a runtime framework to understand and investigate concurrency issues for OpenCL applications. Wang et al. [23] proposed context funneling, which allows kernels from different programs to execute concurrently. None of these works directly addressed the problem of contention caused by multiple concurrently executing kernels in the memory system.

Warp scheduling in GPUs: Narasiman et al. [17] proposed two-level warp scheduler that splits the concurrently executing warps into groups to improve memory latency tolerance. Rogers et al. [22] proposed cache-conscious wavefront scheduling to improve the caching efficiency in GPUs. Gebhart and Johnson et al. [8] proposed a two-level warp scheduling technique that focuses on reducing the energy consumption in GPUs. Jog et al. [12] proposed a series of CTA-aware warp scheduling techniques to reduce cache and memory contention. Kayiran et al. [13] modulated the available thread-level parallelism by intelligent CTA scheduling. Jog et al. [11] proposed prefetch-aware warp scheduling techniques for enhancing GPGPU performance. All these warp scheduling schemes are developed for the scenario when only one kernel is executing at a time. It is not clear how these techniques will perform when multiple kernels are scheduled concurrently. However, in our work, we do not design smart warp scheduling techniques for such scenarios, but do believe that it is an open research issue.

7. CONCLUSIONS AND FUTURE WORK

GPUs are expected to support concurrent execution of multiple kernels – either from the same application or from multiple applications. While this computing paradigm can improve machine utilization when executing applications with limited scalability, the complexity of marshaling multiple kernels introduces key architectural challenges. In this paper we zoomed in on the memory system; we showed that the interactions among memory streams of concurrently executing applications can lead to severe unfairness and suboptimal performance. Furthermore, we showed that the primary reason for these problems is the application-agnostic management of shared resources. For example, the memory scheduler refers to all memory requests as a single request stream and focuses solely on improving the overall DRAM page hit rates.

We argue that in order to overcome these problems, application awareness must be propagated to the memory system. To this end, we proposed a simple augmentation to the current memory system scheduler that schedules memory requests from different applications in a round-robin manner that not only preserves DRAM page hit rates, but also makes sure that co-scheduled memory-intensive applications do not starve other applications for long intervals. Detailed simulation results show that the proposed scheduler delivers superior performance and improves fairness across a wide set of workloads.

Going forward, we plan to explore more sophisticated memory scheduling schemes that exploit knowledge about kernels characteristics to improve scheduling decisions. We also plan to address contention in other shared resources (for e.g., in caches and interconnect) and augment them with application-awareness to provide a complete solution across the entire design.

Acknowledgments

We thank the anonymous reviewers, and the members of the NVIDIA Architecture and HPCL (Penn State) Research Groups for their comments. This work was performed when Adwait Jog was an intern at NVIDIA Research. This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687, #0953246.

References

[1] AMD Radeon R9 290X. http://www.amd.com/us/press-releases/Pages/amd-radeon-r9-290x-2013oct24.aspx.
[2] NVIDIA GRID. http://www.nvidia.com/object/grid-boards.html.
[3] NVIDIA GTX 780-Ti. http://www.nvidia.com/gtx-700-graphics-cards/gtx-780ti/.
[4] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte. The case for GPGPU spatial multitasking. In *HPCA*, 2012.
[5] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.
[6] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.
[7] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel application memory scheduling. In *MICRO*, 2011.
[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 *ISCA*, 2011.
[9] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *HotPar*, 2012.
[10] Hynix. GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0.
[11] 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 *ISCA*, 2013.
[12] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.
[13] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.
[14] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.
[15] D. Kirk and W. W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.
[16] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. *Computer Architecture Letters*, 2012.
[17] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO*, 2011.
[18] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, 2011.
[19] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, 2013.
[20] S. Rixner. Memory controller optimizations for web servers. In *MICRO*, 2004.
[21] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.
[22] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.
[23] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *HPCS*, 2011.
[24] G. Yuan, A. Bakhoda, and T. Aamodt. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *MICRO*, 2009.
[25] W. K. Zuravleff and T. Robinson. Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order. (U.S. Patent Number 5,630,096), Sept. 1997.