Anatomy of GPU Memory System for Multi-Application Execution

Adwait Jog[1,2] Onur Kayiran[2,4] Tuba Kesten[2] Ashutosh Pattnaik[2]
Evgeny Bolotin[3] Niladrish Chatterjee[3] Stephen W. Keckler[3,5]
Mahmut T. Kandemir[2] Chita R. Das[2]

[1]The College of William and Mary [2]The Pennsylvania State University [3]NVIDIA
[4]AMD Research [5]The University of Texas at Austin
adwait@cs.wm.edu, (onur, tzk133, ashutosh, kandemir, das)@cse.psu.edu
(ebolotin, nchatterjee, skeckler)@nvidia.com

ABSTRACT

As GPUs make headway in the computing landscape spanning mobile platforms, supercomputers, cloud and virtual desktop platforms, supporting concurrent execution of multiple applications in GPUs becomes essential for unlocking their full potential. However, unlike CPUs, multi-application execution in GPUs is little explored. In this paper, we study the memory system of GPUs in a concurrently executing multi-application environment. We first present an analytical performance model for many-threaded architectures and show that the common use of misses-per-kilo-instruction (MPKI) as a proxy for performance is not accurate without considering the bandwidth usage of applications. We characterize the memory interference of applications and discuss the limitations of existing memory schedulers in mitigating this interference. We extend the analytical model to multiple applications and identify the key metrics to control various performance metrics. We conduct extensive simulations using an enhanced version of GPGPU-Sim targeted for concurrently executing multiple applications, and show that memory scheduling decisions based on MPKI and bandwidth information are more effective in enhancing system throughput compared to the traditional FR-FCFS and the recently proposed RR FR-FCFS policies.

1. INTRODUCTION

Graphics Processing Units (GPUs) are becoming an inevitable part of heterogeneous computing systems because of their ability to accelerate applications consisting of abundant data-level parallelism. The computing trajectory of GPUs has evolved from traditional graphics rendering to accelerating general purpose and high performance computing applications, and of late to cloud and virtual desktop computing. Two trends have driven this trajectory. First, GPU resources are rapidly growing with each technology generation [2, 4, 6] to provide the workhorse for efficient computation. Second, advances in software and virtualization technology for GPUs such as NVIDIA GRID [3], and OpenStack IaaS framework have made the transition possible.

GPU virtualization is required for concurrent access to the GPU resources by multiple applications, potentially originating from different users. This can be facilitated by spatial as well as temporal allocation of GPU resources. The current NVIDIA GRID [3] and other cloud providers support virtualization by time multiplexing. Spatial resource sharing has yet to evolve because unlike CPUs, traditionally, GPUs were designed to execute only a *single* application at a time. However, it has been shown recently that only executing a single application at a time may not effectively utilize the available computing resources in state-of-the-art GPUs [35], thereby making a compelling case for multi-application execution [22,35]. Thus, supporting multi-application execution is essential both from performance and utility (adoption in cloud environments) perspectives.

Unlike CPU-based architectures, where resource allocation and scheduling for multiple application execution has been studied extensively, only a few recent papers (e.g., [5, 17, 22, 35, 44]) have scratched the issues related to multiple application execution in the context of GPUs. Among them only a few works [22] have addressed the problem of multi-application interference in the GPU memory system. Therefore, many of the design issues are still little understood. In this paper, we focus on the interactions of multiple applications in GPU memory system, and specifically attempt to answer the following questions: (i) *How do we characterize the interactions between multiple applications in the GPU memory system?* (ii) *What are the limitations of traditional application-agnostic FR-FCFS [37,38,46] and RR FR-FCFS scheduling [22] in the context of throughput oriented GPU platforms?*, (iii) *Is it possible to push the performance envelope further with an efficient scheduling mechanism?*, and (iv) *How do we explore the design space and develop analytical performance models to find appropriate knobs for guiding the scheduling decisions?* In this context, this paper makes the following **contributions**:

• Contrary to the common use of misses-per-kilo-instruction (MPKI) as a metric for gauging the memory intensity, and as a proxy for application performance, we show that a model based on **both** MPKI and the achieved DRAM bandwidth information is able to gauge the memory intensity and estimate the performance of GPU applications more accurately.

• We perform a detailed analysis of application characteristics and interactions among multiple applications in GPUs to classify applications and understand the scheduling design space based on this classification.

• We develop a simple analytical model to demonstrate that L2-MPKI and bandwidth utilization can be used as two control knobs to optimize performance metrics: instruction

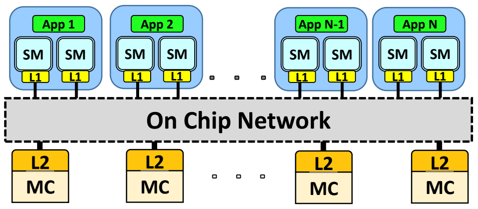


Figure 1: Overview of our baseline architecture capable of executing multiple applications.

throughput (IT) and weighted speedup (WS), respectively. Based on this analytical model, we develop two memory scheduling schemes, ITS and WEIS, which are customized to improve IT and WS, respectively. We show that the proposed solutions are still effective with different core partitioning configurations and scalable for running up to three applications concurrently.

- We qualitatively and quantitatively compare our schemes with the traditional FR-FCFS and the recently proposed round-robin RR FR-FCFS [22] schedulers. Across 25 representative workloads, ITS improves IT by 34% and 8% over FR-FCFS and RR FR-FCFS, respectively; and WEIS improves WS by 10% and 5% over FR-FCFS and RR FR-FCFS, respectively.
- We believe this is the first paper that conducts an in-depth evaluation on GPU memory systems in multi-application environment. In this context, we have developed a GPU Concurrent Application suite (GCA) and a parallel workload simulation framework by extending GPGPU-Sim [8], a cycle accurate GPU simulator. We will open-source the GCA framework for fostering future research.

2. BACKGROUND

2.1 Baseline Architecture

In this paper, we consider a generic NVIDIA-like GPU as our baseline, where multiple cores, also called as streaming multi-processors (SMs)[1], are connected to multiple memory controllers (MCs) via an on-chip network as shown in Figure 1. Each MC is a part of the memory partition that also contains a slice of L2 cache for faster data access. The details of our baseline configuration are shown later in Table 2.

Single Application Scheduling: CUDA uses computational kernels to take advantage of parallel regions in the application. Each application may include multiple kernels. GPUs execute all kernels of an application sequentially, i.e. one kernel at a time. Each kernel is organized as blocks of cooperative thread arrays (CTAs) that are executed in a parallel fashion on the whole GPU. During kernel launch, the CTA scheduler initiates scheduling of the CTAs related to that kernel, and tries to distribute them evenly [8].

Multiple Application Scheduling: In this paper, we simultaneously execute kernels from *different* applications. We use a kernel-to-SMs allocation scheme where we distribute SMs evenly in a spatial manner based on the number of applications as shown in Figure 1. For example, if our GPU consists of 30 SMs that need to be partitioned among two applications, we assign the first 15 SMs to the first application, and the rest of the SMs to the second application. As the focus of this work is memory (not caches), unless otherwise specified, we equally partition SMs and L2 cache across concurrently executing applications. We also evaluate our schemes with two different SM-partitioning techniques (i.e. 10-20 and 20-10) to demonstrate the robustness of our model with respect to core partitioning in Section 8.4.

Memory Scheduling: The most widely used memory scheduler in GPUs is first-ready FCFS (FR-FCFS) [37, 38, 46], which is implemented in the hardware. This scheme is targeted at improving DRAM row hit rates, so request prioritization order is: 1) row-hit requests are prioritized over other requests; then 2) older requests are prioritized over younger ones.

2.2 Evaluation Metrics and Application Suite

Typically, performance of single application is captured by its *instruction throughput (IT)*. When multiple applications execute concurrently, instruction throughput measures the raw machine throughput and is given by $IT = \sum_{i=1}^{N} IPC_i$, where there are N co-running applications, and IPC_i is the number of committed instructions per cycle of the i^{th} application. Note that this metric only considers IPC throughput, without taking fairness into account. In this work, we focus on this metric to evaluate pure machine performance of GPUs without considering the fairness aspect.

For evaluating system throughput, we use *Weighted speedup (WS)*, which indicates how many jobs are executed per unit time: $WS = \sum_{i=1}^{N} SD_i$, where SD_i is the slowdown of i^{th} application given by $SD_i = \frac{IPC_i}{IPC_i^{alone}}$, where IPC_i^{alone} is IPC of i^{th} application when running alone. Assuming there is no constructive interference among applications, the maximum value of WS is equal to the number of applications.

We also show the impact of our schemes on *Harmonic Speedup (HS)* that not only measures system performance, but also has a notion of fairness [28] and is given by, $HS = 1/(\sum_{i=1}^{N} \frac{1}{SD_i})$. Also, the Average Normalized Turn-around Time (ANTT) metric is the reciprocal of HS.

Application Suite: For experimental evaluations, we use a wide range of GPGPU applications implemented in CUDA. These applications are chosen from Rodinia [10], Parboil [41], CUDA SDK [8], and SHOC [11], and are listed in Table 1. In total we study 25 applications.

3. PERFORMANCE CHARACTERIZATION OF MANY-THREADED ARCHITECTURES

In this section, we revisit a performance model for many-threaded architectures proposed by Guz et al. [18], and classify our applications based on this model.

3.1 A Model for Many-threaded Architectures

Recent works have shown that bandwidth is usually the critical bottleneck in many-threaded architectures like GPUs [23, 24, 26, 43]. The considered model [18] shows that performance of many-threaded architectures is directly proportional to the bandwidth that the application receives from DRAM (attained DRAM bandwidth). In this model, application performance is expressed as,

$$P = \frac{BW}{b_{reg} \times r_m \times L_{miss}} \tag{1}$$

[1]In this work, we use "core" and "SM" terms interchangeably.

where

P	=	performance [Operations/second (Ops)]
BW	=	attained DRAM bandwidth [Bytes/second (Bps)]
r_m	=	the ratio of the number of memory instructions to the number of total instructions
b_{reg}	=	the operand size [Bytes]
L_{miss}	=	cache miss rate

This generic model assumes a single memory space, a single-level cache and DRAM. As a result, the miss rate is used for quantifying DRAM accesses. In addition, it ignores the case of multiple memory spaces and the case of scratch-pad usage; e.g. the case where scratch-pad can provide an additional source of bandwidth. Operand size represents the amount of data fetched to/from DRAM on each cache miss, which is equal to the cache line size.

In this model, as the numerator in (1) is BW, performance is directly proportional to the bandwidth attained by the application. We express the denominator as,

$$\begin{aligned} b_{reg} \times r_m \times L_{miss} &= b_{reg} \times \frac{i_{mem}}{i_{tot}} \times \frac{c_{miss}}{c_{tot}} \\ &= b_{reg} \times \underbrace{\frac{c_{miss}}{i_{tot}}}_{\text{MPI}} \times \underbrace{\frac{i_{mem}}{c_{tot}}}_{1} \\ &= \underbrace{b_{reg} \times MPI}_{\substack{\text{Bytes required to transfer from DRAM} \\ \text{for commiting an instruction}}} \end{aligned} \quad (2)$$

where

i_{mem}	=	the number of memory instructions
i_{tot}	=	the number of total instructions
c_{miss}	=	the number of cache misses
c_{tot}	=	the number of cache accesses
MPI	=	the number of cache misses per instruction

Thus, from (1) and (2), we note that performance is directly proportional to the achieved DRAM bandwidth, and is inversely proportional to the size of datum that needs to be fetched to/from DRAM in order to commit an instruction. Since we consider a system with two levels of cache with a constant cache-line size, the overall performance becomes inversely proportional to the number of L2 misses that need to be served for committing one thousand instructions (L2 misses per kilo-instruction (L2MPKI)), as given in (3). In the rest of the paper, we use the term MPKI to represent L2MPKI.

$$P \propto \frac{BW}{MPKI} \quad (3)$$

We validate this model by simulating applications from Table 1 using GPGPU-Sim simulator. Figure 2 shows the observed IPC from the simulator and the calculated IPC values by using Equation (3) and simulated BW and $MPKI$ values. For all 25 applications, the mean absolute relative error (MARE) is 10.3%. We observe that for many (21 out of 25) applications, the MARE is only 4.2%. However, for other 4 applications: `MM`, `HS`, `BP`, and `SAD`, MARE is higher (42.1%), because these applications make extensive use of the software-managed scratchpad memory. As a result, their performance is not only dependent on the achieved DRAM bandwidth, but is also driven by the additional scratchpad bandwidth, which is not captured in our model. For cases that involve usage of a scratch-pad memory, our model underestimates the actual IPC obtained by the application. We conducted a similar analysis on real GPU hardware[2], and Figure 3 shows the absolute relative error for each application. We omit the results for `GUPS`, `MUM` and `QTC`, because we could not execute them faithfully on real hardware. For 22 applications, we observe MARE to be 9.5%.

[2]NVIDIA K20m, CUDA capability 3.5, Driver 6.0

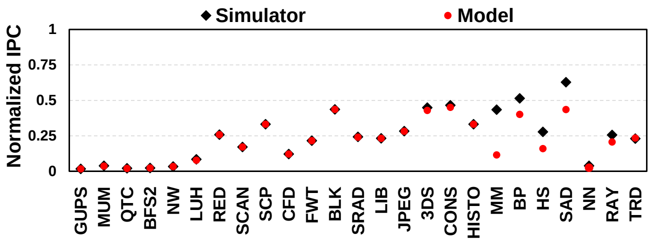


Figure 2: Application performance obtained via simulation and our model. IPC is normalized with respect to the maximum achievable IPC supported by our architecture.

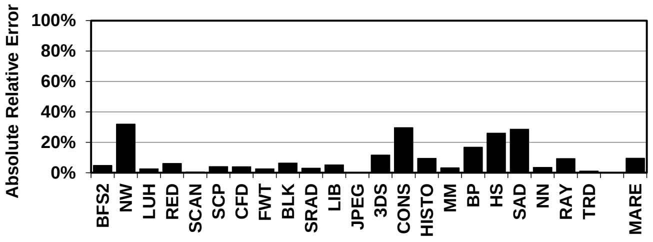


Figure 3: Absolute relative error between IPCs obtained from real hardware (NVIDIA Kepler K20m) and our model.

3.2 Application Characterization

Several previous works (e.g., [12, 27, 28, 30, 36]) have 1) characterized the memory intensity of applications primarily based on their last-level cache MPKI values, and 2) used MPKI as a proxy for performance. However, in this work, we show that considering only MPKI is not enough for the both cases in GPUs.

Table 1 summarizes $MPKI$ and the ratio between attained DRAM bandwidth and peak bandwidth (BW/C) for our applications, where C is the peak memory bandwidth. We list them in the descending order of their $MPKI$. First, we show in Table 1 that only considering $MPKI$ values is not sufficient for estimating the memory intensity of a particular application. It is evident that high $MPKI$ levels may not necessarily lead to a very high DRAM bandwidth utilization, as it is also a function of the inherent compute to bandwidth ratio of a particular application. For example, although `QTC` has the third highest $MPKI$ among our applications, there are 15 other applications in our suite that have higher DRAM bandwidth utilization than `QTC`.

Second, as we showed in Section 3.1, performance is not only dependent on $MPKI$, but also BW. For the applications shown in Table 1, the correlation between $MPKI$ and IPC is only -44.4%. While $MPKI$ is the number of misses that needs to be served to commit 1000 instructions, it lacks the information about the *rate* at which these misses are served by DRAM. In other words, L2-MPKI is a good measure of the bandwidth demand of the application, but performance is a function of *both the demanded and the achieved bandwidth.*

Table 1: Application characteristics: *(A) MPKI*: L2 cache misses per kilo-instructions. *(B) BW/C*: The ratio of attained bandwidth to the peak bandwidth of the system.

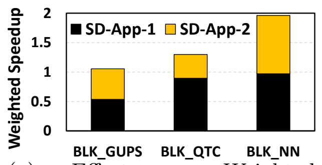
GPGPU Application	Abbr.	MPKI	BW/C (in %)
Random Access	GUPS	57.1	93.0
MUMmerGPU [33]	MUM	22.6	79.7
Quality Threshold Clustering [11]	QTC	7.9	15.3
Breadth-First Search [33]	BFS2	5.3	11.6
Needleman-Wunsch [10]	NW	5.1	16.7
Lulesh [25]	LUH	4.7	35.6
Reduction [11]	RED	2.8	68.8
Exclusive [11] Parallel Prefix Sum	SCAN	2.7	45.0
Scalar Product [33]	SCP	2.7	86.3
Fluid Dynamics [10]	CFD	2.3	27.2
Fast Walsh Transform [33]	FWT	2.2	45.1
BlackScholes	BLK	1.6	67.6
Speckle Reducing Anisotropic Diffusion [10]	SRAD	1.6	36.4
LIBOR Monte Carlo [33]	LIB	1.1	25.4
JPEG Decoding	JPEG	1.1	29.8
3D Stencil	3DS	1.0	42.3
Convolution Separable [33]	CONS	1.0	43.5
2D Histogram [41]	HISTO	0.6	18.8
Matrix Multiplication [41]	MM	0.5	5.1
Backpropogation [10]	BP	0.4	16.5
Hotspot [10]	HS	0.4	6.1
Sum of Absolute Differences [41]	SAD	0.1	5.6
Neural Networks [33]	NN	0.1	0.3
Ray Tracing [33]	RAY	0.1	1.8
Stream Triad [11]	TRD	0.1	1.4

4. ANALYZING MEMORY SYSTEM INTERFERENCE

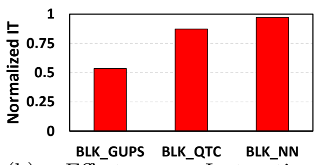
In this section, we build a foundation and draw key observations towards designing a more efficient, application-aware memory scheduler for GPUs. We start with presenting the nature of interference among concurrent applications in GPU memory system, then we discuss the inefficiencies of existing memory schedulers, and finally we draw initial insights for designing a better memory scheduling technique.

4.1 The Problem: Application Interference

When multiple applications are co-scheduled on the same GPU hardware, they interfere at various levels of the memory hierarchy, such as interconnect, caches and memory. As memory system is the critical bottleneck for a large number of GPU applications, explicitly addressing contention issues in the memory system is essential. We find that an uncoordinated allocation of GPU resources, especially memory system resources, can lead to significant performance degradations both in terms of IT and WS. To demonstrate this, consider Figure 4, which shows the impact on WS and IT, when `BLK` is co-scheduled with three different applications (`GUPS`, `QTC`, `NN`). The workloads formed are denoted by `BLK_GUPS`, `BLK_QTC` and `BLK_NN`, respectively. Let us first consider the impact on WS in Figure 4a, where we also show the breakdown of WS in terms of slowdowns (SD) experienced by individual applications. The slowdowns of the applications in the workload are denoted by SD-App-1 and SD-App-2 for the first and the second applications, respectively. Note that when the applications do not interfere with each other, both SD-App-1 and SD-App-2 are equal to 1, leading to a weighted speedup of 2. This figure demonstrates three different memory interference scenarios: (1) in `BLK_GUPS`, both applications slow down each other significantly, (2) in `BLK_QTC`, slowdowns of `BLK` and `QTC` are very different – slowdown of `QTC` being much higher than that of `BLK`, and (3) in `BLK_NN`, slowdown in both applications is negligible. In these different scenarios, the degradation in WS is also very different. We observe significant degradation in WS in the first (90%) and second (54%) cases, whereas, it is negligible (2%) in the third case.



(a) Effect on Weighted Speedup.



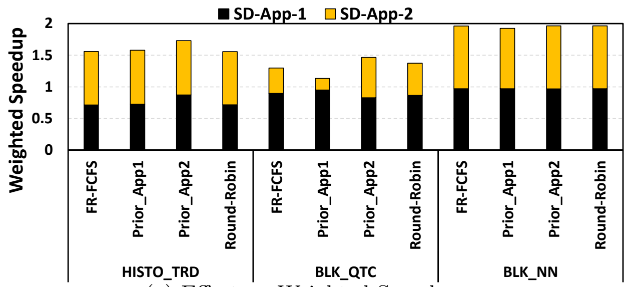
(b) Effect on Instruction Throughput.

Figure 4: Different performance slowdowns obtained when BLK is co-scheduled with three different applications: `GUPS`, `QTC`, and `NN`. Memory scheduling policy is FR-FCFS.

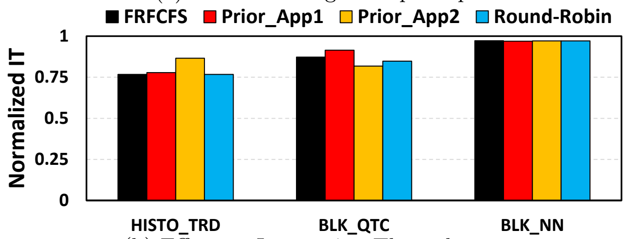
Figure 4b shows IT degradation when `BLK` is co-scheduled with the same three applications vs. the case where `BLK` and other application in the workload are executed sequentially. We observe similar interference trends in `BLK_GUPS` and `BLK_NN`, where in the former case, we observe significant degradation in IT, while the latter exhibits negligible degradation. Interestingly, the IT degradation in `BLK_QTC` is not as significant as compared to its WS degradation. From this discussion, we conclude that *concurrently executing applications are not only susceptible to significant levels of destructive interference in memory, but also the degree with which they impact each other can be strongly dependent on the considered performance metric.*

Analysis: As shown in Section 3, performance of a GPU application is a function of both BW and $MPKI$. As we seek insights on the impact of each metric on performance, let us first discuss the memory bandwidth component. For example, since both `BLK` and `GUPS` have very high bandwidth demands (Table 1), when they are co-scheduled, performance of both applications degrade significantly. We observe exactly the same behaviour in Figure 4a, where both applications do not receive their bandwidth shares for achieving stand-alone performance levels. Similarly, the available DRAM bandwidth is not sufficient for the `BLK_QTC` case, and `BLK` hurts `QTC` performance quite significantly by getting most of the available bandwidth. In the third case with `BLK_NN`, the bandwidth is sufficient for both applications, resulting in negligible performance slowdowns for both. This infers the fact that limited DRAM bandwidth is one of the important reasons of application interference and different application slowdowns.

However, considering only BW of each application would not explain why `BLK_QTC` experiences only a slight degradation in IT. We also need to consider the second parameter that affects performance, $MPKI$, whose values are listed in Table 1. In our example, there is significant difference in $MPKI$ of `BLK` and `QTC` (`BLK` $MPKI$ < `QTC` $MPKI$). From (3), we know that the application with low BW and high $MPKI$ will achieve lower performance levels than the applications with high BW and low $MPKI$. Therefore, in `BLK_QTC`, `BLK` contributes much more towards higher IT than `QTC`. As `BLK` does not slow down significantly (Figure 4a), the total IT reduction is low. This discussion confirms that *both BW and $MPKI$ are key parameters for better understand-*



(a) Effect on Weighted Speedup.



(b) Effect on Instruction Throughput

Figure 5: Different performance slowdowns experienced when different memory scheduling schemes are employed.

ing of the performance characteristics and scheduling considerations for concurrently executing applications in GPUs.

4.2 Limitations of Existing Memory Schedulers

In this section, we discuss the limitations of three memory schedulers. We consider the baseline FR-FCFS [37, 38, 46] scheduler that targets improving DRAM row hit rates, and prioritizes row-hit requests over any other request. In addition, we explore two other schemes a) Prior_App scheduler that *statically* prioritizes requests of only one of the co-scheduled applications, and b) the recently proposed round-robin (RR) FR-FCFS GPU memory scheduler [22] that gives equal priority to all concurrently executed applications in the system without considering their properties. None of these schedulers sacrifice locality, but instead of picking memory requests in FCFS order after servicing row-hit requests (as done in FR-FCFS), Prior_App_i always prefers memory request from i^{th} application, and Round-Robin arbitrates between applications in the round-robin order. Figure 5 shows the effect of these three memory schedulers on weighted speedup and instruction throughput for two of the workloads already shown in Figure 4a (`BLK_QTC` and `BLK_NN`), and an additional workload `HISTO_TRD`.

Limitations of FR-FCFS: When multiple applications are co-scheduled, FR-FCFS still optimizes for row-hit locality and does not consider the individual application properties while making scheduling decisions. Because of such application-unawareness, FCFS nature of the scheduler would allow a high memory demanding application to get a larger bandwidth share, as that application would introduce more requests in memory controller queue. Therefore, as shown in Figure 5, in `BLK_QTC`, we observe that `BLK` gets higher bandwidth share, causing large slowdowns in `QTC`. Moreover, Figure 5 demonstrates that the best performing scheduling strategy for improving either of the the performance metrics is prioritizing one of the applications throughout the entire execution.

Limitations of Prior_App_i: We observe in `HISTO_TRD` that prioritizing `TRD` over `HISTO` provides the best IT and WS among all the considered scheduling strategies. However, in `BLK_QTC`, prioritizing one application over another does not improve both the performance metrics. In `BLK_NN`, since both the applications attain their uncontested bandwidth demands, prioritizing one over another does not impact performance. Even though prioritizing one application over another provides the best result, the challenge is to determine which application to prioritize. One way of doing so is to profile the workload and employ a static priority mechanism throughout the execution. However, such strategy is often hard to realize. Another mechanism can be to switch priorities between applications during run-time, which is similar to RR FR-FCFS [22].

Limitations of RR FR-FCFS: As discussed above, in order to optimize WS and IT in `BLK_QTC`, we should prioritize different applications. Since RR switches priorities between these two applications, it achieves a balance between improving both metrics. However, in `HISTO_TRD`, although employing RR mechanism leads to a slightly better IT and WS over FR-FCFS, it is far from the case where we prioritize `TRD`.

Based on the above discussion, we make two key observations that guide us in developing an application-conscious scheduling strategy.

Observation 1: Prioritizing lower $MPKI$ applications improves IT: In the workloads where we observe significant slowdowns in either of the applications, prioritizing the application with lower $MPKI$ improves IT. For example, in `HISTO_TRD`, `TRD` has lower $MPKI$ than `HISTO` and therefore, Prior_App_2 yields better IT. Similarly, in `BLK_QTC`, `BLK` has lower $MPKI$ than `QTC`, thus Prior_App_1 provides better IT.

Observation 2: Prioritizing lower BW applications improves WS: In the workloads where we observe significant slowdowns in either of the applications, prioritizing the application with lower BW improves WS. For example, in `HISTO_TRD`, `TRD` has lower BW than `HISTO` and therefore Prior_App_2 yields better WS. Similarly, in `BLK_QTC`, `QTC` has lower BW than `BLK`, thus Prior_App_2 provides better WS.

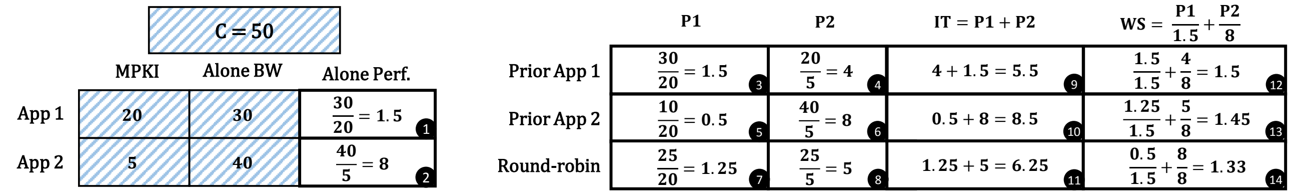
5. A PERFORMANCE MODEL FOR CONCURRENTLY EXECUTING APPLICATIONS

In order to provide a theoretical background for the observations we discussed for optimizing IT and WS in Section 4.2, we extend the model discussed in Section 3 to two applications, and analyze the effect of concurrent execution on the performance metrics *when memory bandwidth is the system bottleneck*. We also show how this model guides us in developing different memory scheduling algorithms targeted for optimizing each metric separately, and explain our findings based on our model by examples.

The notations we use in our model are given below:

P_i^{alone}	=	performance of application i when it runs alone
BW_i^{alone}	=	bandwidth attained by application i when it runs alone
P_i	=	performance of application i
BW_i	=	bandwidth attained by application i
C	=	peak bandwidth of the system
ϵ	=	infinitesimal bandwidth given to an application
$MPKI_i$	=	$MPKI$ of application i

When applications run alone, they cannot attain more bandwidth than that is available in the system, thus



C = 50

	MPKI	Alone BW	Alone Perf.
App 1	20	30	$\frac{30}{20} = 1.5$ ❶
App 2	5	40	$\frac{40}{5} = 8$ ❷

	P1	P2	IT = P1 + P2	$\text{WS} = \frac{\text{P1}}{1.5} + \frac{\text{P2}}{8}$
Prior App 1	$\frac{30}{20} = 1.5$ ❸	$\frac{20}{5} = 4$ ❹	$4 + 1.5 = 5.5$ ❾	$\frac{1.5}{1.5} + \frac{4}{8} = 1.5$ ⓬
Prior App 2	$\frac{10}{20} = 0.5$ ❺	$\frac{40}{5} = 8$ ❻	$0.5 + 8 = 8.5$ ❿	$\frac{1.25}{1.5} + \frac{5}{8} = 1.45$ ⓭
Round-robin	$\frac{25}{20} = 1.25$ ❼	$\frac{25}{5} = 5$ ❽	$1.25 + 5 = 6.25$ ⓫	$\frac{0.5}{1.5} + \frac{8}{8} = 1.33$ ⓮

Figure 6: An illustrative example showing IT and WS for two applications running together. The shaded boxes represent system and application properties. The peak memory bandwidth is 50 units. Application 1 and 2 use 30 and 40 units bandwidth, respectively, when they execute alone. Their $MPKI$s are 20 and 5, respectively.

$BW_i^{alone} \leq C \ \forall i$; and similarly for concurrent execution, the cumulative bandwidth consumption cannot exceed the peak bandwidth, thus $\sum_{i=1}^{N} BW_i \leq C$. Also, we have $P_i, BW_i, MPKI_i \geq 0 \ \forall i$.

Let us analyze the case when two applications are executed concurrently[3]. Assuming that performance is limited by the memory bandwidth, an application cannot consume more bandwidth without getting more share from the other application's bandwidth. Thus, in a two-application scenario, $BW_1 + BW_2 = C$, assuming no wastage of bandwidth is incurred.

We assume that, at time $t = t_0$, we have $P_i \propto \frac{BW_i}{MPKI_i}$ $\forall i$; and at $t = t_1$ where $t_1 > t_0$, we give an additional ϵ bandwidth to the first application by taking it from the other. Thus, at $t = t_1$, we have $P_1' \propto \frac{BW_1+\epsilon}{MPKI_1}$, and $P_2' \propto \frac{BW_2-\epsilon}{MPKI_2}$, where P_i' is the performance of application i at $t = t_1$.

5.1 Analyzing Instruction Throughput

In order to have higher IT at $t = t_1$ compared to $t = t_0$,

$$P_1' + P_2' > P_1 + P_2 \tag{4}$$

$$\frac{BW_1 + \epsilon}{MPKI_1} + \frac{BW_2 - \epsilon}{MPKI_2} > \frac{BW_1}{MPKI_1} + \frac{BW_2}{MPKI_2} \tag{5}$$

Simplifying (5) yields,

$$\epsilon(MPKI_2 - MPKI_1) > 0 \tag{6}$$

$$\implies MPKI_2 > MPKI_1, \text{if } \epsilon > 0 \tag{7}$$

$$MPKI_1 > MPKI_2, \text{if } \epsilon < 0 \tag{8}$$

From (7) and (8), we find that *we should prioritize the application with lower $MPKI$ in order to optimize IT.*

Illustrative Example: Figure 6 shows an example of the optimal scheduling strategy for maximizing IT. In this example, we consider three different scheduling strategies: 1) prioritizing the first application, 2) prioritizing the second application, and 3) the RR scheduler. When these applications run alone, based on their BW and $MPKI$ values, we can say that the first and the second applications achieve performances of 1.5 (❶) and 8 units (❷), respectively, based on (3). If these applications are co-scheduled on the GPU, and if we prioritize the first application throughout the execution, that application will get 30 units of bandwidth, since it demands 30 units when it runs alone. We assume that $MPKI$ does not change significantly during different executions, as we do not do any cache-related optimization in this work for preserving simplicity. Because the first application's $MPKI$ is 20, its performance will be 1.5 units (❸). The remaining 20 units of bandwidth will be used by the second application, as the peak bandwidth is 50 units ($C = 50$). Because its $MPKI$ is 5, its performance will be 4 units (❹). Similarly, if we prioritize the second application, it will get 40 units of bandwidth as it demands 40 units when running alone, and the first application will use the remaining 10 units. Thus, the performances of the first and the second applications will be 0.5 (❺) and 8 units (❻), respectively. Also, if we employ the round-robin scheduler, assuming both applications have similar row-buffer localities, they get the same share from the available bandwidth, which is 25 units, leading to 1.25 (❼) and 5 (❽) units of performance for the first and the second applications, respectively. Based on these individual application performance values calculated using our model, we show that IT (the sum of individual IPCs) of this workload is 5.5 (❾), 8.5 (❿), and 6.25 units (⓫), when we prioritize the first application, prioritize the second application, and employ RR scheduler, respectively. These results are consistent with our model, which suggests that prioritizing the application that has lower $MPKI$ would provide the best IT. Intuitively, as per (3), the same BW provided for the application with lower $MPKI$, which is the second application in this example, translates to higher IPC. Thus, prioritizing the application with lower $MPKI$ provides higher IT for the system.

We observe in Figure 5b that prioritizing the application with lower $MPKI$ results in higher IT for `HISTO_TRD` and `BLK_QTC`. Since the application interference in `BLK_NN` is not significant, it does not benefit from prioritization.

[3]Our model can be easily extended to analyze more than two applications. We show the scalability of our model for three applications in Section 8.4.

5.2 Analyzing Weighted Speedup

In order to have higher WS at $t = t_1$ compared to $t = t_0$,

$$\frac{P_1'}{P_1^{alone}} + \frac{P_2'}{P_2^{alone}} > \frac{P_1}{P_1^{alone}} + \frac{P_2}{P_2^{alone}} \tag{9}$$

$$\frac{\frac{BW_1+\epsilon}{MPKI_1}}{\frac{BW_1^{alone}}{MPKI_1}} + \frac{\frac{BW_2-\epsilon}{MPKI_2}}{\frac{BW_2^{alone}}{MPKI_2}} > \frac{\frac{BW_1}{MPKI_1}}{\frac{BW_1^{alone}}{MPKI_1}} + \frac{\frac{BW_2}{MPKI_2}}{\frac{BW_2^{alone}}{MPKI_2}} \tag{10}$$

Simplifying (10) yields,

$$\epsilon(BW_2^{alone} - BW_1^{alone}) > 0 \tag{11}$$

$$\implies BW_2^{alone} > BW_1^{alone}, \text{if } \epsilon > 0 \tag{12}$$

$$BW_1^{alone} > BW_2^{alone}, \text{if } \epsilon < 0 \tag{13}$$

From (12) and (13), we find that *we should prioritize the application with lower BW^{alone} to optimize weighted speedup.*

Illustrative Example: We continue the example shown in Figure 6 with the optimal scheduling strategy for maximizing WS. We obtain $WS = 1.5$ (⓬), 1.33 (⓭), and 1.45 units (⓮) if we prioritize the first application, the second application, and employ RR scheduler, respectively. As discussed above, prioritizing the second application provides the best IT. However, we also observe that prioritizing the application with the lowest BW^{alone}, which is the first application in this example, yields the best WS, which is consistent with our model. Intuitively, the same amount of bandwidth provided for the application with lower BW^{alone}, the first application in this example, translates to lower performance degradation over its uncontested performance for that application. Since WS is the sum of slowdowns, prioritizing the application with lower BW^{alone} provides higher WS for the system.

We observe in Figure 5a that prioritizing the application with lower BW^{alone} results in higher WS for `HISTO_TRD` and `BLK_QTC`. `BLK_NN` is not a bandwidth limited workload, thus, does not benefit from prioritization.

However, the problem with the approach that prioritizes the application with lower BW^{alone} is that, it is difficult to obtain BW^{alone} of an application without offline profiling, as pointed out by Subramanian et al. [42]. They also propose an approximate method to obtain alone performance of an application in a multiple-application environment during run-time. However, doing so requires halting the execution of one of the applications to approximately calculate the alone performance of the other application, which might cause drop in WS. Also, it does not completely eliminate the application interference. Furthermore, this sampling has to be done frequently to capture execution phases with completely different behaviours. Thus, instead of approximating P_i^{alone} or BW_i^{alone}, we slightly change our WS optimization condition, which leads to a solution that is much easier to implement and employ during run-time. Our approximation does not use alone performance of an application; instead, compares P_i' with P_i. Mathematically, we have,

$$\frac{P_1'}{P_1} + \frac{P_2'}{P_2} > \frac{P_1}{P_1} + \frac{P_2}{P_2} \quad (14)$$

$$\frac{BW_1 + \epsilon}{MPKI_1}\frac{MPKI_1}{BW_1} + \frac{BW_2 - \epsilon}{MPKI_2}\frac{MPKI_2}{BW_2} > 2 \quad (15)$$

Simplifying (15) yields,

$$\epsilon(BW_2 - BW_1) > 0 \quad (16)$$

$$\implies BW_2 > BW_1, \text{if } \epsilon > 0 \quad (17)$$

$$BW_1 > BW_2, \text{if } \epsilon < 0 \quad (18)$$

From (17) and (18), we find that *we can prioritize the application with lower BW to improve relative weighted speedup.*

We will later demonstrate in Section 8 that such approximation leads to better weighted speedup. This is also consistent with the observation made by Kim et al. [27] that preferring application with the least attained bandwidth can improve weighted speedup.

Discussion: We showed in Figure 6 that optimizing both IT and WS using the same memory scheduling strategy might not be possible. We also showed a similar scenario in Figure 5 where `BLK_QTC` prefers a different application to be prioritized in order to achieve the best IT or WS. The key reason behind this is the properties of the applications that form the workload. In workloads, where the same application has the lower $MPKI$ and the lower BW^{alone}, that application can be prioritized to optimize both IT and WS. However, in workloads, where one application has lower $MPKI$ but the other demands less bandwidth, then the optimal scheduling strategy for improving our performance metrics is different. In such scenarios, RR achieves a good balance between IT and WS, because it gives equal priorities to both applications (assuming both applications have similar row-buffer localities). However, in scenarios where prioritizing only one application is optimal for both IT and WS, RR would be far from optimal in terms of performance.

6. MECHANISM AND IMPLEMENTATION DETAILS

We provided two key observations in Section 4.2, and presented a theoretical background for them in Section 5. Based on these observations, we propose two memory scheduling schemes: a) Instruction Throughput Targeted Scheme (ITS), and b) Weighted Speedup Targeted Scheme (WEIS).

1) Instruction Throughput Targeted Scheme (ITS): This scheme aims to improve IT based on our observation that the application having lower $MPKI$ should be prioritized. In order to determine the application with lower $MPKI$, we first periodically (every 1024 cycles[4]) calculate two metrics during run-time: 1) the number of L2 misses for each application locally at each memory partition, and 2) the number of instructions committed by each application. Then, the information regarding the committed instructions is propagated to the MCs. We calculate $MPKI$ of all the applications locally at each MC, using an arithmetic unit. Then, by using a comparator, we determine the application with the lowest $MPKI$, and prioritize the memory requests generated by that application in the MC.

2) Weighted Speedup Targeted Scheme (WEIS): This scheme aims to improve WS based on our observation that the application having lower BW should be prioritized. In order to determine the application with lower BW, we periodically calculate the amount of data transferred over the DRAM bus for each application locally at each memory partition. Then, by using a comparator, we determine the application with the lowest BW, and prioritize the memory requests generated by that application in the MC.

Implementation of the Priority Mechanism: Our priority mechanism takes advantage of the already existing FR-FCFS memory scheduler implementation. However, after serving the requests that generate DRAM row buffer hits, instead of picking request in the FCFS order, we pick the oldest request from the highest priority application. When there is no request from the highest priority application in the MC queue, we pick the oldest request originating from the application with the next highest priority.

7. INFRASTRUCTURE AND EVALUATION METHODOLOGY

Infrastructure: Most of the prior GPU research (e.g. [7, 9, 21, 23, 24, 26]) is focused on improving the performance of a single GPU application, and is evaluated based on the benchmarks originating from different application suites (Section 3.2). However, to investigate the research issues in the context of multiple applications, these applications need to be concurrently executed on the same GPU platform. This is a non-trivial task because it involves building

[4]We also used three other sampling size windows (256, 1024, 2048) cycles. The difference in overall average performance is less than 1%, implying that sampling window size does not have a significant impact on our design.

Table 2: Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.1 [16] for full list.

Core Features	1400MHz core clock, 30 cores (streaming multi-processors), SIMT width = 32 (16 × 2), Greedy-then-oldest first (GTO) dual warp scheduler [34]
Resources / Core [16, 39, 40]	16KB shared memory, 16KB register file, Max. 1536 threads (48 warps, 32 threads/warp)
Private Caches / Core [16, 39, 40]	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
Shared L2 Cache	16-way 128 KB/memory channel (768KB in total), 128B cache block size
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Memory Model	6 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling (256 max. requests/MC), 8 DRAM-banks/MC, 4 bank-groups/MC, 924 MHz memory clock, 88.8GB/sec peak memory bandwidth Global linear address space is interleaved among partitions in chunks of 256 bytes [15] Reads and writes are handled with equal priority [8, 16] Hynix GDDR5 Timing [19], t_{CL} = 12, t_{RP} = 12, t_{RC} = 40, t_{RAS} = 28, t_{CCD} = 2, t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12
Interconnect [16]	1 crossbar/direction (30 cores, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

a framework that can launch existing CUDA applications in parallel without significant changes to the source code. In order to do so, we develop a new framework, called GPU concurrent application framework (GCA). This framework takes advantage of CUDA *streams*. A stream is defined as a series of memory operations and kernel launches that are required to execute sequentially, however, different streams can be executed in parallel. Our GCA framework creates a separate stream for each application, and issues all its associated commands to the stream. As many of the legacy CUDA codes use synchronous (e.g., *cudaMemcpy()*) memory transfer operations, our framework does source code modifications to change them to asynchronous CUDA API calls (e.g., *cudaMemcpyAsync()*). To ensure correct execution of multiple streams, our framework also adds appropriate synchronization constructs (e.g., *cudaStreamSynchronize()*) to the source code at correct places. After these steps are performed, GCA is ready to execute multiple streams/applications either on real GPU hardware or on the simulator. In this paper, we use GCA to concurrently execute workloads on GPGPU-Sim, which is already capable of concurrently running multiple CUDA streams. As a part of initial package, our suite contains 300 ($\binom{25}{2}$) 2-application workloads and 2300 ($\binom{25}{3}$) 3-application workloads. Also, the initial GCA package includes guidelines to add a new CUDA code to the suite. We will open source the GCA package for public use for fostering future research.

Evaluation Methodology: We simulate both two- and three-application workloads on GPGPU-Sim [8], a cycle-accurate GPU simulator. Table 2 provides the details of simulation configuration. GCA framework launches CUDA applications on GPGPU-Sim and executes until the point where all the applications complete at least once. To achieve this, GCA framework relaunches the faster running application(s) until the slowest application completes its execution. We collect the statistics of individual applications when they finish their respective executions such that amount of work done by individual applications across different runs is consistent. This methodology is consistent with the prior works [35]. We only simulate application kernels, and do not have a performance simulation for the data transfer between CPU and GPU. Our DRAM contention model is the same that comes with the default GPGPU-Sim distribution, which is validated across many workloads [8].

Workload Classification: We evaluate 100 two-application workloads and classify them based on two criteria. The first classification is based on the $MPKI$ difference between the applications in the workload. If this difference is greater than 10, the workload belongs to

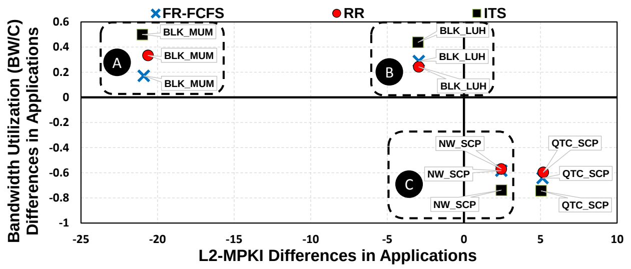


Figure 7: The effect of FR-FCFS, RR, and ITS on $BW_1 - BW_2$ and $MPKI_1 - MPKI_2$.

Class-A-MPKI. If it is less than 1, the workload belongs to Class-C-MPKI, otherwise it belongs to Class-B-MPKI. The second classification is based on the BW/C (bandwidth-utilization) difference between the applications in the workload. If this difference is greater than 50%, the workload belongs to Class-A-BW. If it is less than 25%, the workload belongs to Class-C-BW, otherwise it belongs to Class-B-BW. The intuition behind this classification method is that the workloads with high $MPKI$ and high BW difference are more likely to benefit from ITS and WEIS, respectively.

8. EXPERIMENTAL RESULTS

In this section, we evaluate five memory scheduling mechanisms: 1) the baseline FR-FCFS, 2) the recently proposed RR FR-FCFS [22], 3) ITS, 4) WEIS, 5) a static mechanism that always prioritizes the lowest $MPKI$ application in the workload, Prior_App_Low_MPKI, and 6) a static mechanism that always prioritizes the application in the workload with the lowest BW^{alone}, Prior_App_Low_BW. Note that the above static mechanisms require offline profiling that are difficult to employ during run-time. We use these static schemes as comparison points for ITS and WEIS, respectively. We use equal core partitioning across SMs for the applications in a workload, and show sensitivity to different core partitioning schemes in Section 8.4. We use geometric mean (GM) to report average performance results. For each proposed scheme, we first demonstrate how effective the algorithm is in modulating the bandwidth given to each application, and then we show the performance results.

8.1 Evaluation of ITS

How ITS Works: Figure 7 shows the effect of FR-FCFS, RR FR-FCFS, and ITS on four representative workloads. The x-axis shows $MPKI_1 - MPKI_2$, and the y-axis shows $BW_1 - BW_2$.

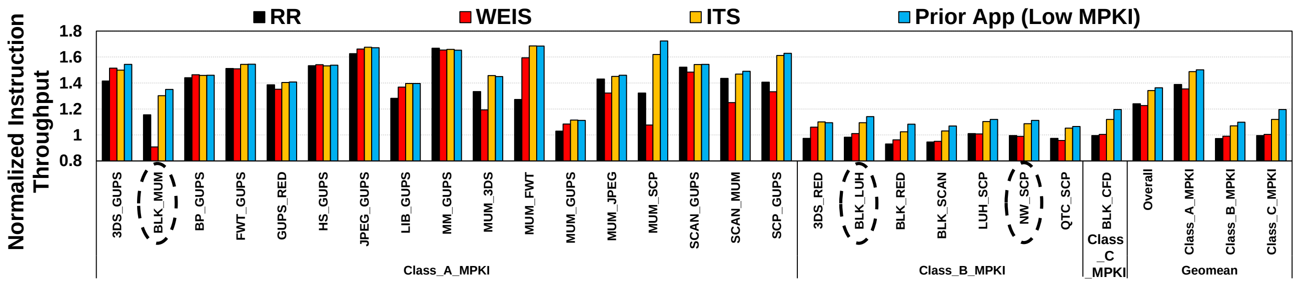


Figure 8: IT results normalized with respect to FR-FCFS for 25 representative workloads.

In BLK_MUM which is shown in Ⓐ, since MUM has higher $MPKI$ than BLK, and BLK attains higher BW than MUM, all the points inside Ⓐ are in the second quadrant. ITS prioritizes BLK due to its relatively lower $MPKI$, thus, the difference between BW attained by BLK and MUM increases with respect to FR-FCFS. We observe an interesting case in RR. Since BLK already attains higher bandwidth with FR-FCFS, we would expect MUM to find more opportunity to utilize DRAM with RR. However, the RR mechanism employed by Jog et al. [22] preserves row-locality while scheduling requests. Therefore, this mechanism, although expected to give equal share of bandwidth to both applications, provides more opportunity to the application with higher row-locality for scheduling its requests. In other words, RR provides the applications with an equal opportunity to activate rows, resulting in the application with higher row-locality to schedule more requests due to their differences between the number of requests served per active row-buffer. The exact phenomenon is observed in Ⓐ with RR, where BLK has 4× higher row-locality than that of MUM, leading to RR giving BLK even higher opportunity to schedule its requests compared to FR-FCFS. However, ITS unilaterally prefers BLK due to its consistently lower $MPKI$.

In BLK_LUH, which is shown in Ⓑ, we observe very similar trends as in Ⓐ. However, as opposed to Ⓐ, RR reduces the gap between BW achieved by LUH and BLK, since both applications have similar row-localities. In NW_SCP which is shown in Ⓒ, since NW has higher $MPKI$ than SCP, and SCP attains higher bandwidth than NW, all the points inside Ⓒ are in the fourth quadrant. ITS prioritizes SCP due to its relatively lower $MPKI$, thus, SCP achieves even more BW compared to FR-FCFS. We observe almost the same behavior with QTC_SCP as well. Note that, $MPKI$ values of the applications in the workload do not change across schemes, as it is an application property and each application has its own L2 cache partition.

ITS Performance: Figure 8 shows the instruction throughput of RR, WEIS, ITS, and Prior_App_Low_MPKI normalized with respect to FR-FCFS, using 25 representative workloads that span across $MPKI$-based workload classes, chosen from our pool of 100 workloads. We also show the average IT of these workloads including the individual GM for each workload class. As expected, in BLK_MUM previously shown in Ⓐ (Figure 7), IT improves by 15% with RR, and by 30% with ITS. We observe that, employing WEIS provides improvements over FR-FCFS, but results in slightly lower average performance than RR. It is expected, because WEIS is not targeted to optimize IT. With ITS, we observe 34% and 8% average IT improvements over FR-FCFS and RR, respectively, across 25 workloads. These numbers are 49% and 7% for Class-A-MPKI applications, because the

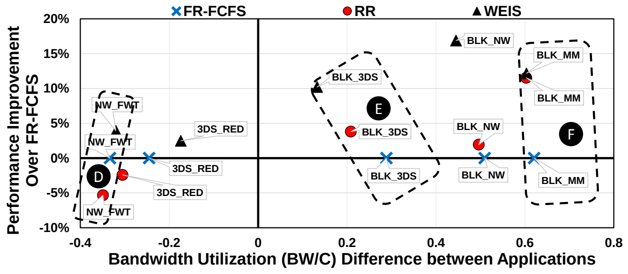


Figure 9: Effect of FR-FCFS, RR, and WEIS on WS and $BW_1 - BW_2$.

workloads that have applications with strikingly different $MPKI$s are more likely to benefit with ITS over FR-FCFS. Class-B-MPKI and Class-C-MPKI also gain moderate performance improvements, by 7% and 12% over FR-FCFS, respectively. As we have shown in Figure 7, $MPKI$ does not change significantly. Thus, dynamism of ITS does not provide extra IT benefits over Prior_App_Low_MPKI.

8.2 Evaluation of WEIS

How WEIS Works: Figure 9 shows the effect of FR-FCFS, RR, and ITS on five representative workloads. The x-axis shows $BW_1 - BW_2$, and the y-axis shows the normalized WS improvement over FR-FCFS. WEIS attempts to reduce the difference between BW attained by the applications, and therefore in the figure, we expect WEIS to push the workloads towards the y-axis. Also, as it is expected to improve WS, it also pushes the workload upwards. In NW_FWT which is shown in Ⓓ, BW of FWT is higher than NW. WEIS prefers NW as it attained lower BW, which pushes this workloads upwards and towards y-axis. The RR mechanism degrades WS for NW_FWT because of similar reasons related to row-locality as pointed earlier (FWT has 13× higher row-locality than NW).

In BLK_3DS (Ⓔ) and BLK_MM (Ⓕ) both RR and WEIS push the workload towards y-axis along with improving WS. We observe that the trends in both RR and WEIS are similar in 3DS_RED and NW_FWT, and also in BLK_NW and BLK_3DS.

WEIS Performance: Figure 10 shows WS of RR, WEIS, ITS, and Prior_App_Low_BW normalized with respect to FR-FCFS, using 25 representative workloads that span across BW-based workload classes, chosen from our pool of 100 workloads. We also show the average WS of these workloads including the individual GM for each workload class. We observe that, employing ITS provides improvements over FR-FCFS, but results in lower average performance than RR. This is expected, because ITS is not targeted to optimize WS. With WEIS, we observe 10% and 5% average WS improvements over FR-FCFS and RR, respectively, across 25 workloads. These numbers are 14%

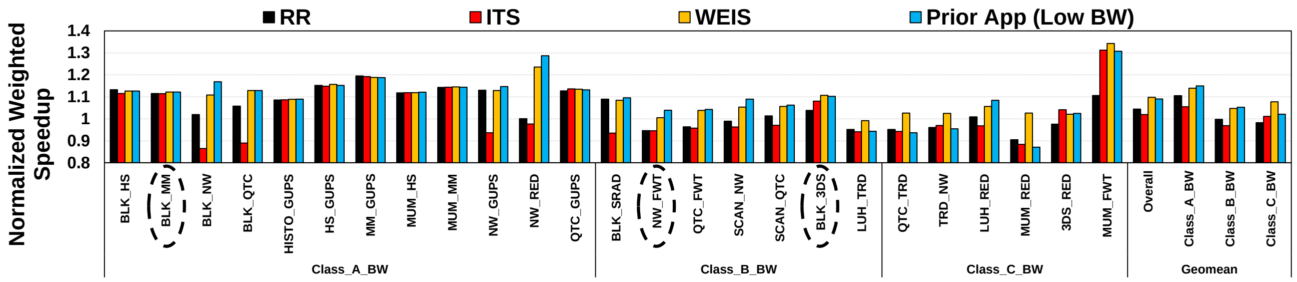
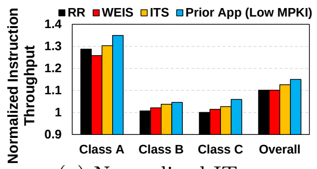


Figure 10: WS results normalized with respect to FR-FCFS for 25 representative workloads.


(a) Normalized IT.

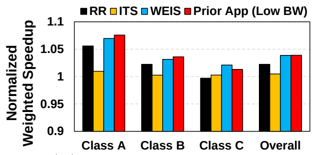

(b) Normalized WS.

Figure 11: Summary IT and WS results for 100 workloads, normalized with respect to FR-FCFS.

and 3% for Class-A-BW applications, because the workloads that have applications with strikingly different BW are more likely to benefit with WEIS, compared to FR-FCFS. Class-B-BW and Class-C-BW also gain performance improvements, by 5% and 8% over FR-FCFS, respectively. In Class-C-BW workloads, WEIS performs much better than Prior_App_Low_BW. This is because the average BW difference is not significant, and it is more likely that the same application does not achieve consistently lower bandwidth than the other application. Therefore, prioritizing an application unilaterally like Prior_App_Low_BW does may lead to sub-optimal performance.

8.3 Performance Summary

We evaluate ITS and WEIS for 100 workloads and we observe in Figure 11 that our conclusions from previous discussions hold true for a wide range of workloads. In Figure 12, we report Harmonic Speedup (HS) for 100 workloads in order to gauge WEIS with a balanced metric for performance as well as fairness [28]. Across all classes of workloads, we consistently observe better HS compared of FR-FCFS and RR. On average, RR and WEIS achieve 4% and 8% higher HS over FR-FCFS.

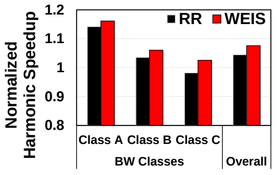
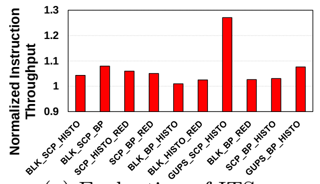


Figure 12: HS results for 100 workloads normalized with respect to FR-FCFS.

8.4 Scalability Analysis

Application Scalability: We evaluate ITS and WEIS in the scenario when three applications are executed concurrently. We observe that the impact of our schemes is even higher, as there is significant increase in memory interference among three applications. Figure 14 shows normalized IT and WS improvements with ITS and WEIS, respectively for 10 workloads. We observe significant IT improvement (27%) in `GUPS_SCP_HISTO`, as ITS prefers `HISTO` because of its significantly lower $MPKI$ than other applications in the workload. For WEIS, we also observe similar trends as discussed before.

Core Partitioning: We evaluate three core partitioning configurations: (10,20), (20,10), and the baseline (15,15). Figure 13 shows normalized IT improvements of ITS for `JPEG_GUPS`, over FR-FCFS when it is used in their respective configurations. We observe in all three configurations that the improvements in `JPEG_GUPS` are significant. However, if fewer cores (ITS (20,10)) are assigned to `GUPS`, which is a very high memory demanding application, the negative interference effect on `JPEG` is reduced. Therefore, the relative IT improvements in ITS (20,10) is lower than ITS (15, 15). In the case of ITS (10, 20), the alone IPC of `JPEG` is lower as `JPEG` is assigned to fewer cores. This leads to lower scope in `JPEG` IPC improvements compared to the baseline ITS (15, 15) case. These results indicate that although core partitioning mechanisms affect the magnitude of interference, the problem still remains significant. We leave orchestrated memory and core resource partitioning schemes as a part of future work.


(a) Evaluation of ITS.

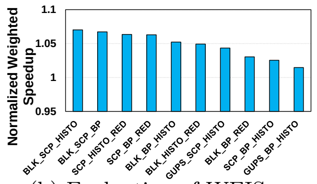

(b) Evaluation of WEIS.

Figure 14: Evaluation of ITS and WEIS with three GPU applications.

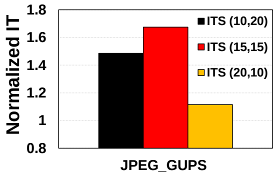


Figure 13: Core partitioning results.

9. RELATED WORK

To the best of our knowledge, this is the first work that analyzes the interactions of multiple applications in GPU memory system, via both experiments as well as a mathematical model.

GPU and SoC Memory Scheduling: Prior work on memory scheduling for GPUs has dealt with a single application context only. Yuan et al. [45] proposed an arbitration mechanism in NoC to restore the lost row-buffer locality to enable a simple in-order DRAM memory scheduler. Lakshminarayana et al. [29] explored a DRAM scheduling policy that essentially chooses between Shortest Job First (SJF) and FR-FCFS [38,46]. Chatterjee et al. [9] proposed a warp-aware memory scheduling mechanism that reduces the DRAM latency divergence in a warp by avoiding the interleaved servicing of memory requests from different warps. The benefits from the above schedulers are orthogonal to our schemes and some of these mechanisms can be adapted as secondary arbitration criteria between requests for the currently prioritized application in our scheduler. In the SoC space, Jeong et al. [20] proposed allowing the GPU to consume only the required bandwidth to maintain a certain real-time QoS-level for graphics applications. Ausavarungnirun et al. [7] proposed a memory scheduling technique for CPU-GPU architectures. However, the overriding motivations for such prior work is to obtain the lowest possible latency for CPU requests without degrading the bandwidth utilization of the channel.

CPU Memory Schedulers: The impact of memory scheduling on multicore CPU systems has been a topic of significant interest in recent years [1,14,27,28,31,32]. Ebrahimi et al. [13] proposed parallel application memory scheduling, where they explicitly managed inter-thread memory interference for improving performance. The Thread Cluster Memory Scheduler (TCM) [28] is particularly relevant because not only did it advocate prioritizing latency-sensitive applications, it identified that the main source of unfairness is the interference between different bandwidth intensive applications. To improve performance and fairness, TCM ranks the bandwidth-intensive threads based on their relative bank-level parallelism and row-buffer locality, and periodically shuffles the priority of the threads. The TCM technique is an advancement over the ATLAS [27], PARBS [32], and STFM [31] mechanisms that do not distinguish between bandwidth-intensive threads while improving performance and fairness.

We share the same objectives as the CPU schedulers like TCM, but the motivations and considerations behind our proposed schedulers, as well as the implementation and derived insight are significantly different. First, prior CPU memory schedulers concentrated only on single-threaded or modestly multi-threaded/multi-programmed workloads, while we demonstrate the benefits of our schemes for multiple, massively-threaded applications running on a fundamentally different architecture (SIMT). Second, the analysis in Sec. 3 establishes a different set of metrics from TCM, *viz.* $MPKI$ and attained bandwidth, to guide the memory scheduling at the application level (as opposed to single threads as in TCM). This is partly due to the use of TLP in GPU programs to hide memory latency as opposed to ILP and MLP in CPUs. Third, in contrast to earlier papers, we demonstrate that the same scheduler can not achieve the best of both aggregate throughput and fairness. Consequently, a practical implementation can use runtime settings to choose between high throughput and fairness-optimized scheduling based on application domains (e.g., high aggregate throughput in HPC applications vs QoS guarantees in virtualized GPUs). The final differentiator between TCM and our mechanisms is complexity. TCM needs to track *each thread's* MLP, bank-level parallelism (BLP), and row-buffer locality (RBL), and requires an expensive insertion sort-like procedure to shuffle the ranks of high-MLP applications. In contrast, we only require the L2 MPKI and currently sustained bandwidth information for *each application*, and a few simple comparisons in each time quanta. Scaling TCM's policies for the many thousands of concurrent threads in a GPU would be challenging in GDDR5 MC that has to support multi-gigabit command issue rates.

Concurrent execution of multiple applications on GPUs: Adriaens et al. [5] proposed spatial partitioning of SM resources across concurrent applications. They presented a variety of heuristics for dividing the SM resources across applications. Pai et al. [35] proposed elastic kernels that allow a fine-grained control over their resource usage. None of these works addressed the problem of contention in the memory system. Moreover, we show that memory interference problem remains significant regardless of SM-partitioning mechanisms, and we believe our proposed schemes are complementary to core resource partitioning techniques. Gregg et al. [17] presented KernelMerge, a runtime framework to understand and investigate concurrency issues for OpenCL applications. Wang et al. [44] proposed context funneling, which allows kernels from different programs to execute concurrently. Our work presents a new GCA framework that consists of large number of CUDA workloads and also provides flexibility to add new CUDA codes in the framework without much effort.

10. CONCLUSIONS

We present an in-depth analysis of GPU memory system in a multiple-application domain. We show that co-scheduled applications can significantly interfere in the GPU memory system leading to significant loss in overall performance. To address this problem, we developed an analytical model that indicates that L2-MPKI and attained bandwidth are the two knobs that can be used to drive memory scheduling decisions for achieving better performance.

11. REFERENCES

[1] 3rd JILP Workshop on Computer Architecture Competitions (Memory Scheduling Championship). `http://www.cs.utah.edu/~rajeev/jwac12/`.

[2] AMD Radeon R9 290X. `http://www.amd.com/us/press-releases/Pages/amd-radeon-r9-290x-2013oct24.aspx`.

[3] NVIDIA GRID. `http://www.nvidia.com/object/grid-boards.html`.

[4] NVIDIA GTX 780-Ti. `http://www.nvidia.com/gtx-700-graphics-cards/gtx-780ti/`.

[5] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte. The case for GPGPU spatial multitasking. In *HPCA*, 2012.

[6] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. *SIGARCH Comput. Archit. News*.

[7] 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.

[8] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[9] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *SC*, 2014.

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

[11] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S.

Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU*, 2010.

[12] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Aérgia: exploiting packet latency slack in on-chip networks. In *ACM SIGARCH Computer Architecture News*. ACM, 2010.

[13] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, 2011.

[14] S. Ghose, H. Lee, and J. F. Martínez. Improving memory scheduling via processor-side load criticality information. In *ISCA*, 2013.

[15] GPGPU-Sim v3.2.1. Address mapping.

[16] GPGPU-Sim v3.2.1. GTX 480 Configuration.

[17] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *HotPar*, 2012.

[18] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Comput. Archit. Lett.*

[19] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0.

[20] 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 the 49th Annual Design Automation Conference*, pages 850–855. ACM, 2012.

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

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

[23] 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.

[24] 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.

[25] I. Karlin, A. Bhatele, J. Keasler, B. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *IPDPS*, 2013.

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

[27] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

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

[29] 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.

[30] X. Lin and R. Balasubramonian. Refining the utility metric for utility-based cache partitioning. *Proc. WDDD*, 2011.

[31] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.

[32] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.

[33] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.

[34] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, 2011.

[35] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, 2013.

[36] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.

[37] S. Rixner. Memory Controller Optimizations for Web Servers. In *MICRO*, 2004.

[38] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

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

[40] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-Aware Warp Scheduling. In *MICRO*, 2013.

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

[42] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, 2013.

[43] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. GDM: Device Memory Management for Gpgpu Computing. In *SIGMETRICS*, 2014.

[44] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *HPCS*, 2011.

[45] G. Yuan, A. Bakhoda, and T. Aamodt. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *MICRO*, 2009.

[46] 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.