2018 IEEE International Symposium on High Performance Computer Architecture

Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management

Haonan Wang*, Fan Luo*, Mohamed Ibrahim*, Onur Kayiran†, and Adwait Jog*
*College of William and Mary †Advanced Micro Devices, Inc.
Email: {hwang07,fluo,maibrahim}@email.wm.edu, onur.kayiran@amd.com, adwait@cs.wm.edu

***Abstract*—**
Managing the thread-level parallelism (TLP) of GPGPU applications by limiting it to a certain degree is known to be effective in improving the overall performance. However, we find that such prior techniques can lead to sub-optimal system throughput and fairness when two or more applications are co-scheduled on the same GPU. It is because they attempt to maximize the performance of individual applications in isolation, ultimately allowing each application to take a disproportionate amount of shared resources. This leads to high contention in shared cache and memory. To address this problem, we propose new application-aware TLP management techniques for a multi-application execution environment such that all co-scheduled applications can make good and judicious use of all the shared resources. For measuring such use, we propose an application-level utility metric, called *effective bandwidth*, which accounts for two runtime metrics: attained DRAM bandwidth and cache miss rates. We find that maximizing the total effective bandwidth and doing so in a balanced fashion across all co-located applications can significantly improve the system throughput and fairness. Instead of exhaustively searching across all the different combinations of TLP configurations that achieve these goals, we find that a significant amount of overhead can be reduced by taking advantage of the trends, which we call *patterns*, in the way application's effective bandwidth changes with different TLP combinations. Our proposed pattern-based TLP management mechanisms improve the system throughput and fairness by 20% and 2×, respectively, over a baseline where each application executes with a TLP configuration that provides the best performance when it executes alone.

***Index Terms*—Bandwidth Management, Fairness, GPUs**

I. Introduction

Graphics Processing Units (GPUs) are becoming an inevitable part of most computing systems because of their ability to provide orders of magnitude faster and energy-efficient execution for many general-purpose applications [11], [13], [36], [38], [39], [41], [46], [51], [52], [56], [60]. With each new generation of GPUs, peak memory bandwidth and throughput are growing at a steady pace [1], [62], and it is expected to continue as technology scales and new emerging high-bandwidth memories become mainstream. To keep these growing compute and memory resources highly utilized, GPU multi-programming has recently received significant attention [2], [17], [24], [25], [34], [35], [37], [43], [44], [45], [58], [61], [65], [67]. The idea of co-locating two or more kernels[1] (originating from the same or different applications) has been shown to be beneficial in terms of both GPU resource utilization and throughput [24], [25], [43].

One of the major roadblocks in achieving the maximum benefits of multi-application execution is the difficulty to design mechanisms that can efficiently and effectively manage the application interference in the shared caches and the main-memory. Several researchers have proposed different architectural mechanisms (e.g., novel resource allocation [7], [8], [9], cache replacement [47], [48], and memory scheduling [24], [25]), both in CPU and GPU domains, to address the negative effects of the shared-resource application interference on the system throughput and fairness. In fact, a recent surge of works [27], [28], [30], [31], [50] considered the problem of managing inter-thread cache/memory interference even for the single GPU application execution. In particular, the techniques that find the optimal thread-level parallelism (TLP) of a GPGPU application were found to be very effective in improving the GPU performance both for cache and memory-intensive applications [30], [50]. The key idea behind these techniques is to exploit the observation that executing a GPGPU application with the maximum possible TLP does not necessarily result in the highest performance. This is because as the number of concurrently executing threads increases, the contention for cache space and memory bandwidth also increases, which can lead to sub-optimal performance. Therefore, many of the TLP management techniques proposed to limit the TLP via limiting the number of concurrently executing warps (or wavefronts[2]) to a particular value.

Inspired by the benefits of such TLP management techniques, this paper delves into the design of new TLP management techniques that can significantly improve the system throughput and fairness by modulating the TLP of each concurrently executing application. To understand the scope of TLP modulation in the context of multi-application execution, we analyzed three different scenarios. First, both applications are executed with their respective best-performing TLP (bestTLP), which are found by statically profiling each application separately by running it alone on the GPU. Note that these individual best TLP configurations can also be effectively calculated using previously proposed runtime mechanisms (e.g., DynCTA [30], CCWS [50]). We call this multi-application TLP combination as ++bestTLP. Second, both applications are executed with their respective maximum possible TLP (maxTLP). We call this multi-application TLP combination as ++maxTLP. Third, both applications are executed with TLP such that they collectively achieve the highest Weighted Speedup (WS) (or Fairness Index

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

[2]A wavefront is a group of threads, and the granularity at which hardware scheduling is performed. It is referred to as a "warp" in NVIDIA terminology.

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00030

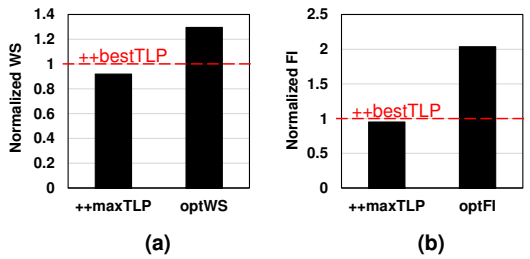


Fig. 1: Weighted Speedup (WS) and Fairness Index (FI) for `BFS2_FFT`. Evaluation methodology is described in Section II.

(FI))[3] and defined as optWS (or optFI).

Figure 1 shows the WS and FI when `BFS2` and `FFT` are executed concurrently under these three aforementioned scenarios[4]. The results are normalized to ++bestTLP. We find that there is a significant difference in WS and FI between optimal (optWS and optFI) and ++bestTLP combinations, which suggests that blindly using the bestTLP configuration for each application in the context of multi-application execution is a sub-optimal choice. It is because each application in the ++bestTLP or ++maxTLP scenario consumes disproportionate amounts of shared resources as it assumes no other application is co-scheduled. That leads to high cache and memory contention.

Contributions. To our knowledge, this is the first work that performs a detailed analysis of the TLP management techniques in the context of multi-application execution in GPUs and shows that new TLP management techniques, if developed carefully, can significantly boost the system throughput and fairness. In this context, our *goal* is to develop techniques that can find the optimal TLP combination that allows a judicious and good use of all the available shared resources (thereby reducing cache and memory bandwidth contention, and improving WS and FI). To measure such use, we propose a new metric, *effective bandwidth (EB)*, which calculates the effective shared resource usage for each application considering its private and shared cache miss rates and memory bandwidth consumption. We find that a TLP combination that maximizes the total effective bandwidth across all co-located applications while providing a good balance of individual applications' effective bandwidth leads to high system throughput (WS) and fairness (FI). Instead of incurring the high overheads of an exhaustive search across all the different combinations of TLP configurations that achieve these goals, we propose pattern-based searching (PBS) that cuts down a significant amount of overheads by taking advantage of the trends (which we call *patterns*) in the way application's effective bandwidth changes with different TLP combinations.

Results. Our newly proposed PBS schemes improve the system throughput (WS) and fairness (FI) by 20% and 2×, respectively over ++bestTLP, and 10% and 1.44×, respectively over the recently proposed TLP modulation and cache bypassing scheme (Mod+Bypass [35]) for multi-application execution for GPUs. Also, our PBS schemes are within 3% and 6% of the *optimal* TLP combinations: optWS and optFI, respectively for the evaluated 50 two-application workloads.

[3]We find this combination by profiling 64 different combinations of TLP and picking the one that provides the best WS (or FI).

[4]BFS2_FFT is one of the representative workloads, which demonstrates the scope of the problem. Other workloads are discussed in Section VI.

II. Background and Evaluation Methodology

A. Baseline Architecture

We consider a generic baseline GPU architecture consisting of multiple cores (also called as compute units (CUs) or streaming multiprocessors (SMs) in AMD and NVIDIA terminology, respectively.). Each core has a private L1 cache. An on-chip interconnect connects the cores to the memory controllers (MCs). Each MC is attached to a slice of L2 cache. We evaluate our proposed techniques on MAFIA [25], a GPGPU-Sim [4] based framework that can execute two or more applications concurrently. The memory performance model is validated across several GPGPU workloads on an NVIDIA K20m GPU [4], [25]. The key parameters of the GPU (Table I) are faithfully simulated.

TABLE I: Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.2 [16] for the complete list.

Core Features	1400MHz core clock, 30 cores, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32684 registers Max. 1536 threads (48 warps, 32 threads/warp)
L1 Caches / Core	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
L2 Cache	16-way 256 KB/memory channel (1536 KB 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 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock Global linear address space is interleaved among partitions in chunks of 256 bytes [15] Hynix GDDR5 Timing [22], t_{CL} = 12, t_{RP} = 12, t_{RAS} = 28, t_{CCD} = 2, t_{RCD} = 12, t_{RRD} = 6
Interconnect	1 crossbar/direction (30 cores, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

Multi-application execution. Recent GPUs by AMD [3] and NVIDIA [42] support the execution of multiple tasks/kernels. This advancement led to a large body of work in GPU multiprogramming [2], [17], [24], [25], [34], [35], [37], [43], [44], [58], [65], [67]. Execution of multiple tasks can potentially increase GPU utilization and throughput [2], [24], [43], [59], [64]. While it is possible to execute different independent kernels from the same application concurrently, in this paper, we execute different applications simultaneously. To understand how these applications interfere in the shared memory system, each application is mapped to an exclusive set of cores and allowed to use resources beyond the cores (e.g., L2, DRAM). We allocate an equal number of cores to each concurrently executing application. Sensitivity to different core and L2 cache partitioning techniques is discussed in Section VI-D.

TLP configurations. We assume that different TLP configurations are implemented at the warp granularity via statically or dynamically limiting the number of actively executing warps [50]. Table II lists different TLP configurations that are evaluated in the paper. The maximum value of TLP is 24 as the total number of possible warps on a core is 48 and there are two warp schedulers per core. The baseline GPU uses the best-performing TLP (bestTLP) when it executes only one application at a time.

B. Evaluation Methodology

All evaluated metrics are summarized in Table III.

TABLE II: List of evaluated TLP configurations.

Acronym	Description
maxTLP	Single application is executed with the maximum possible value of TLP.
++maxTLP	Two or more applications are executed concurrently with their own respective maxTLP configurations.
bestTLP	Single application is executed with the best-performing TLP.
++bestTLP	Two or more applications are executed concurrently with their own respective bestTLP configurations.
DynCTA	Single application is executed with DynCTA.
++DynCTA	Two or more applications are executed concurrently with each one using DynCTA.
optWS	Two or more applications are executed concurrently with their own TLP configurations such that Weighted-speedup (WS) is maximized.
optFI	Two or more applications are executed concurrently with their own TLP configurations such that Fairness Index (FI) is maximized.
optHS	Two or more applications are executed concurrently with their own TLP configurations such that Harmonic Weighted-speedup (HS) is maximized.

TABLE III: List of evaluated metrics.

Acronym	Description
SD	Slowdown. SD = IPC-Shared/IPC-bestTLP.
WS	Weighted Speedup. WS = SD-1 + SD-2 .
FI	Fairness Index. FI = Min(SD-1/SD-2, SD-2/SD-1)
HS	Harmonic Speedup. HS = 1/(1/SD-1 + 1/SD-2).
BW	Attained Bandwidth from Main Memory.
CMR	Combined Miss Rate (MR). CMR = L1MR $\times$ L2MR.
EB	Effective Bandwidth. EB = BW/CMR.
EB-WS	EB-based Weighted Speedup. EB-WS = EB-1 + EB-2 .
EB-FI	EB-based Fairness Index. EB-FI = Min(EB-1/EB-2, EB-2/EB-1).
EB-HS	EB-based Harmonic Speedup. EB-HS = 1/(1/EB-1 + 1/EB-2).

Performance- and Fairness-related Metrics. We report *Weighted Speedup* (WS) [43], [55] and Fairness Index (FI) [25] to measure system throughput and fairness (imbalance of performance slowdowns), respectively. Both metrics are based on individual application slowdowns (SDs) in the workload, where SD is defined as the ratio of performance (IPC) achieved in the multi-programmed environment (IPC-Shared) to the case when it runs alone on the same set of cores with bestTLP (IPC-Alone). The maximum value of WS is equal to the number of applications in the workload assuming there is no constructive interference among applications. An FI value of 1 indicates a completely fair system. We also report Harmonic Weighted Speedup (HS), which provides a balanced notion of both system throughput and fairness in the system [33]. In this paper, we refer to all these metrics as SD-based metrics.

Auxiliary Metrics. We consider *Attained Bandwidth* (BW), which is defined as the amount of DRAM bandwidth that is useful for the application (i.e., the useful data transferred over the DRAM interface) normalized to the theoretical peak value of the DRAM bandwidth. We also consider *Combined Miss Rate* (CMR), which is defined as the product of L1 and L2 miss rates. Note that BW and L1/L2 miss rates are separately calculated for each application even in the multi-application scenario. The *Effective Bandwidth* (EB) of an application is the ratio of BW to CMR. It gauges the rate of data delivery to cores by considering how the bandwidth achieved from DRAM is amplified by the caches (e.g., a miss rate of 50% effectively doubles the bandwidth delivered.). We append the application ID (or application's abbreviation (Table IV)) to the end of these metrics to denote per-application metrics (e.g., CMR-1 is the combined miss rate for application 1 and EB-BLK is the effective bandwidth for CUDA Blackscholes Application).

EB-based Metrics. In addition to standard SD-based metrics that we finally report, our proposed techniques take advantage of runtime *EB-based metrics*. These metrics are calculated in a similar fashion as SD-based metrics with the difference that they use EB instead of SD. For example, EB-WS is defined as the sum of EB-1 and EB-2. More details are in Table III and in upcoming sections.

TABLE IV: GPGPU application characteristics: *(A) IPC@bestTLP: The value of IPC when the application executes with bestTLP, (B) EB@bestTLP: The value of the effective bandwidth when the application executes with bestTLP, (C) Group information: Each application is categorized into one of the four groups (G1-G4) based on their individual EB values.*

Abbr.	IPC	EB	Group	Abbr.	IPC	EB	Group
LUD [5]	40	0.13	G1	LIB [40]	211	0.93	G2
NW [5]	31	0.21	G1	LUH [29]	87	1.08	G2
HISTO [57]	471	0.29	G1	SRAD [5]	229	1.19	G3
SAD [57]	651	0.31	G1	CONS [40]	397	1.35	G3
QTC [6]	26	0.59	G2	FWT [40]	195	1.41	G3
RED [6]	180	0.70	G2	BP [5]	580	1.42	G3
SCAN [6]	151	0.72	G2	CFD [5]	95	1.49	G3
BLK [40]	457	0.79	G2	TRD [6]	238	1.67	G3
HS [5]	578	0.79	G2	FFT [57]	261	1.77	G4
SC [5]	173	0.80	G2	BFS2 [40]	18	1.78	G4
SCP [40]	307	0.85	G2	3DS	457	2.19	G4
GUPS	9	0.87	G2	LPS [40]	410	2.20	G4
JPEG [40]	330	0.92	G2	RAY [40]	328	3.12	G4

Application suites. For our evaluations, we use a wide range of GPGPU applications with diverse memory behavior in terms of cache miss rates and memory bandwidth. These applications are chosen from Rodinia [5], Parboil [57], CUDA SDK [4], and SHOC [6] based on their effective bandwidth (EB) values such that there is a good spread (from low to high – see Table IV). In total, we study 50 two-application workloads (spanning 26 single applications) that exhibit the problem of multi-application cache/memory interference.

III. Analyzing Application Resource Consumption

In this section, we first discuss the effects of TLP on various single-application metrics followed by a discussion on succinctly quantifying those effects.

A. Understanding Effects of TLP on Resource Consumption

GPU applications achieve significant speedups in performance by exploiting high TLP. Therefore, GPU memory has to serve a large number of memory requests originating from many warps concurrently executing across different GPU cores. Consequently, memory bandwidth can easily become the most critical performance bottleneck for many GPGPU applications [14], [18], [25], [28], [30], [49], [62], [63]. Many prior works have proposed to address this bottleneck by improving the bandwidth utilization and/or by effectively using both private and shared caches in GPUs via modulating the available TLP [30], [50]. These modulation techniques strive to improve performance via finding the level of TLP such that it is neither too low so as not to under-utilize the on/off-chip resources nor too high so as not to cause too much contention in caches and memory leading to poor cache miss rates and row-buffer locality, respectively. To understand this further, consider Figure 2 (a–c), which shows the impact of the change in TLP (i.e., different levels of TLP) for `BFS2` on IPC, BW, and CMR. These metrics are normalized to that of bestTLP (best performing TLP for `BFS2` is 4). We observe

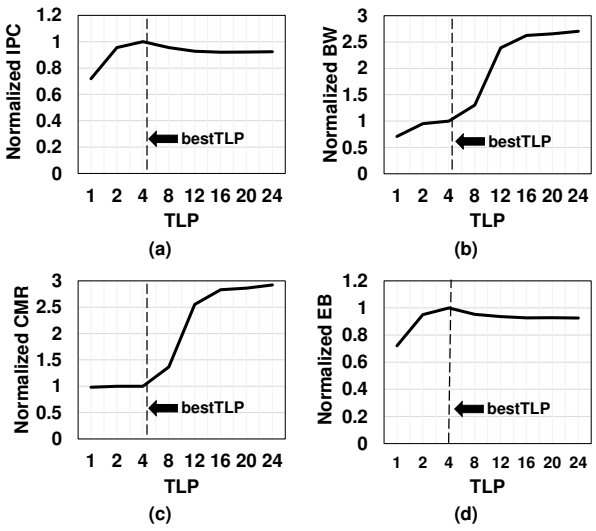


Fig. 2: Effect of TLP on performance and other metrics for BFS2.

that with the initial increase in TLP, both BW and IPC start to increase rapidly. However, at higher TLP, the increase in CMR starts to negate the benefits of high BW, ultimately leading to decrease in performance. For example, when TLP limit increases from 4 to 24, BW increases by 2.7×, but CMR also increases by 2.9× leading to drop in performance. In such cases, the increase in TLP not only hampers performance but also consumes unnecessary memory bandwidth. *In summary, we conclude that the changes in performance with different TLP configurations are directly related to changes in cache miss-rate and memory bandwidth resource consumption.*

B. Quantifying Resource Consumption

To measure such resource consumption via a single combined metric, we introduce a new metric called as *effective bandwidth (EB)*. It is defined as the ratio of bandwidth to miss rate, and is calculated based on the level of hierarchy under consideration as depicted in Figure 3. For example, the value of EB observed by L1 (Ⓑ) is defined as the ratio of BW (Ⓐ) to the L2 miss rate. Similarly, the value of EB observed by the core (Ⓒ) is defined as the ratio of EB observed by L1 (Ⓑ) to the L1 miss rate. This value is also equivalent to the ratio of BW (Ⓐ) to CMR. The EB observed by the core essentially measures how well the DRAM bandwidth is utilized. It also considers the usefulness of the caches in amplifying the performance impact of the attained DRAM bandwidth, where the amplification is based on the combined miss rate. If the CMR is 1, it implies that caches are not useful and cores will obtain the same return bandwidth that is attained from the DRAM. Therefore, EB is equal to BW for cache insensitive applications (e.g., BLK). On the other hand, a lower CMR would allow cores to obtain more return bandwidth than what is attained from the DRAM, which is the case for cache-sensitive applications (e.g., BFS2). In an ideal case, when combined miss rate is zero, the effective bandwidth observed by the core would be equal to the L1 cache bandwidth of the GPU system[5].

[5]It is under the assumption that the return packets from the memory system do not bypass the L1 cache.

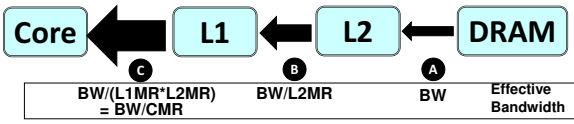


Fig. 3: Effective Bandwidth at different levels of the hierarchy. For brevity, we show only one core (with attached L1 cache) and one L2 cache partition.

Connecting back to Figure 2, we observe that effective bandwidth observed by the core (Ⓒ) and performance closely follow each other (Figure 2(d)) with changes in TLP. This concludes that the impact of changes in TLP on performance can be accurately estimated by directly measuring only the changes in EB, *without* the need to consider any architecture-independent parameters such as compute-to-memory ratio of an application. Although we demonstrate the validity of these conclusions for BFS2, we verified that these conclusions hold true for all the considered applications[6] listed in Table IV.

To substantiate the conclusions analytically, we revisit our prior work [25], which showed that GPU performance (IPC) is proportional to the ratio of BW to L2 misses per instruction (MPI),

$$IPC \propto \frac{BW}{L2MPI} \tag{1}$$

which in turn is proportional to the ratio of BW to $r_m \times CMR$, where r_m is the ratio of memory instructions to the total number of instructions. r_m is an application-level property[7]. Therefore, IPC is proportional to the ratio of EB to r_m, where

$$IPC \propto \frac{BW}{r_m \times L2MR \times L1MR} \propto \frac{BW}{r_m \times CMR} \propto \frac{EB}{r_m} \tag{2}$$

We conclude that EB is able to effectively measure the good and judicious use of cache and memory bandwidth resources and is optimal at the bestTLP configuration.

IV. Motivation and Goals

As the total shared resources are limited, understanding how these resources should be allocated to the concurrent applications for maximizing system throughput and fairness is a challenging problem. To this end, we consider the TLP of *each* application as a knob to control its shared resource allocation. Although previously proposed TLP modulation techniques (e.g., DynCTA [30] and CCWS [50]) have been shown to be effective in optimizing TLP for single-application execution scenarios, they rely only on different kinds of *per-core heuristics* (e.g., latency tolerance, IPC, cache/memory contention) and do not consider the shared resource consumption of co-scheduled applications. In other words, each application under such TLP configurations (including bestTLP) attempts to maximize its *own* effective bandwidth, ultimately taking disproportionate amount of shared resources. This causes too much contention

[6]Applications that make heavy use of the software-managed scratchpad memory observe higher EB at the cores due to additional bandwidth from the scratchpad memory. Because the scratchpad memory is not susceptible to contention due to high TLP in our evaluation setup, our calculations do not consider the bandwidth provided by the scratchpad to the core.

[7]Arithmetic intensity (i.e., ratio between compute to memory instructions) of an application is equal to (1-r_m)/r_m.

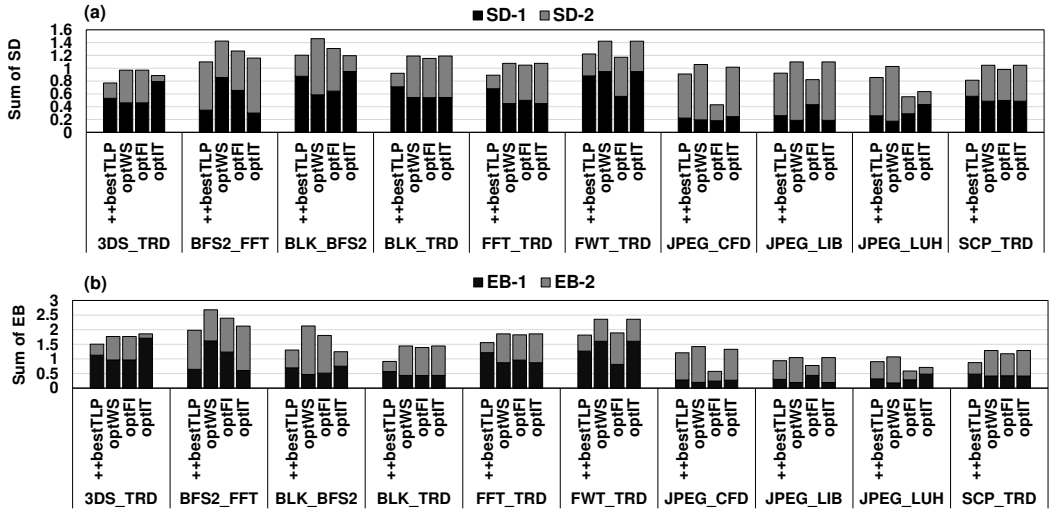


Fig. 4: Effect of different TLP combinations on application: a) slowdown and b) effective bandwidth.

in caches and memory, ultimately hampering the system-wide metrics such as system throughput and fairness.

To understand this further, consider Figure 4(a), which shows the WS of 10 representative workloads under ++bestTLP and opt TLP combinations[8]. WS for each two-application workload is split into its respective slowdowns for each application (SD-1 and SD-2). We find that there is a significant gap between ++bestTLP and optWS for all workloads. For example, in `BFS2_FFT` and `BLK_BFS2`, this difference is 29% and 21%, respectively. In terms of fairness, the gap is up to 2× (observed from the imbalance between SD values in ++bestTLP compared to balanced values in optFI) in `BFS2_FFT`. *We conclude that new TLP management techniques are needed to close this system throughput and fairness gap.*

Analysis of Weighted Speedup. We find that a TLP management scheme that optimizes for EB-based metrics is useful in improving system performance and fairness. To understand this analytically, we first focus on system throughput (weighted speedup) via equations 3, 4, and 5. First, let us define IPC alone ratio (IPC_{AR}) and EB alone ratio (EB_{AR}) of two applications (App-1 and App-2) when each of them separately execute alone on the GPU:

$$\begin{aligned} IPC_{AR} &= \frac{IPC_{Alone-1}}{IPC_{Alone-2}} \\ EB_{AR} &= \frac{EB_{Alone-1}}{EB_{Alone-2}} \end{aligned} \tag{3}$$

Next, as WS is the sum of slowdowns of co-scheduled applications, we derive the following Equation:

$$\begin{aligned} WS &= \frac{IPC_{Shared-1}}{IPC_{Alone-1}} + \frac{IPC_{Shared-2}}{IPC_{Alone-2}} \\ WS &\propto IPC_{Shared-1} + IPC_{Shared-2} \times IPC_{AR} \end{aligned} \tag{4}$$

Finally, with the help of Equation 2,

$$WS \propto EB_{Shared-1} + EB_{Shared-2} \times EB_{AR} \tag{5}$$

We observe that WS is a function of instruction throughput (sum of IPCs of individual applications) and also a function of EB-WS (sum of EBs of individual applications). However, maximizing IT or EB-WS will lead to sub-optimal WS, if IPC_{AR} and EB_{AR} are much greater than 1. This is due to the bias caused by alone ratios (IPC_{AR} and EB_{AR}) towards one of the co-scheduled applications. On average across all possible two-application workloads formed using the evaluated 26 applications, we find that EB_{AR} is much lower than IPC_{AR}, as shown in Figure 5[9]. Therefore, we choose EB to optimize system-wide metrics.

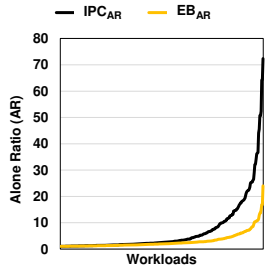


Fig. 5: IPC_{AR} **vs.** EB_{AR}

To substantiate this claim quantitatively, Figure 4(b) shows EB-WS for each workload along with its respective EBs for each application (EB-1 and EB-2). We make the following two observations:

- **Observation 1:** The TLP combination that provides the highest sum of EB (EB-WS) provides also the highest system throughput (WS). This trend is present in almost all evaluated workloads (a few exceptions are discussed in Section VI). We find this observation interesting as it means that optimizing for EB-WS is likely to improve WS (SD-based metric) as discussed earlier via Equation 5. Also, EB-WS metric *does not* incorporate any alone-application information making it easier to calculate directly in a multi-application environment.
- **Observation 2:** The TLP combination (optIT) that provides the highest instruction throughput (IT) (i.e., sum of IPCs across all the concurrent applications) *does not* always provide the highest WS and FI (e.g., in `BFS2_FFT`, `BLK_BFS2`). It implies that a mechanism that attempts to maximize IT may not be optimal to improve system throughput as analytically demonstrated earlier.

[8]The opt combinations are chosen via an exhaustive search of 64 different TLP combinations.

[9]As the alone ratio bias can be towards any one of the co-scheduled applications, we show max(M1/M2, M2/M1), where M is IPC_{AR} or EB_{AR}.

Analysis of Fairness. Extending the above discussion for fairness, we also find that EB-FI correlates well with the SD-based FI (i.e., differences in SDs in a workload is correlated with those of EBs.). Therefore, a careful balance of effective bandwidth allocation among co-scheduled applications can lead to higher fairness in the system (as demonstrated by optFI in Figure 4(b)). However, there are a few outliers (e.g., BLK_TRD) (i.e., the difference between EB-1 and EB-2 is much larger than that of SD-1 and SD-2 breakdowns). The main reason behind these outliers is EB_{AR}, which can still be larger than one (Figure 5) leading to a bias towards one of the applications. To reduce the outliers and increase the correlation between EB-FI and SD-FI, we appropriately scale the EB with the alone EB information of each application. These *scaling-factors* either can be supplied by the user or can be calculated at runtime. In the former case, each application uses the average value of alone EB for the *group* it belongs to (see Table IV). In the latter case, each application uses the value of EB when it executes alone and uses bestTLP. As we cannot get this information unless we halt the other co-running applications, we approximate it by executing the co-runners with the least amount of TLP (i.e., 1) so that they induce the least amount of interference possible. Note that in our evaluated workloads, we did not find the necessity of using these scaling factors while optimizing WS because of the limited number of outliers (Section VI). However, we do use them to further optimize fairness and harmonic weighted speedup.

In summary, we conclude that maximizing the total effective bandwidth (EB-WS) for all the co-runners is important for improving system throughput (WS). Further, a better balance between the effective bandwidth (determined by EB-FI) of the co-scheduled applications is required for higher fairness (FI).

V. Pattern-based Searching (PBS)

In this section, we provide details on the proposed TLP management techniques for multi-application execution followed by the implementation details and hardware overheads.

A. Overview

Our goal is to find the TLP combinations that would optimize different EB-based metrics. A naive method to achieve this goal is to periodically take samples for all the possible TLP combinations over the course of workload execution and ultimately choose the combination that satisfies the optimization criteria. However, that would incur significant runtime overheads in terms of performance. Instead of high-overhead naive searching, we take advantage of the following guidelines and *patterns* to minimize the search space for optimizing the EB-based metrics.

Guideline-1. The EB-based metrics are sub-optimal when a particular TLP combination leads to under-utilization of resources (e.g., DRAM bandwidth). Therefore, for obtaining the optimal system throughput, *it is important to choose a TLP combination that does not under-utilize the shared resources.*

Guideline-2. When increasing an application's TLP level, its EB starts to drop only when the increase in its BW can no longer compensate for the increase in its CMR (i.e., EB at its inflection point). Therefore, *it is important to choose a TLP combination that would not overwhelm resources as it is likely to cause sharp drops in one or all applications' EB, leading to inferior system throughput and fairness.*

Patterns. In all our evaluated workloads, we find that when resources in the system are sufficiently utilized, distinct inflection points emerge in EB-based metrics. These inflection points tend to appear consistently at the same TLP level of an application, regardless of the TLP levels of the other co-running application. We name this consistency of the inflection points as *patterns*. Moreover, the sharpest drop in EB-based metrics is usually attributed to one of the co-running applications, namely the *critical* application.

High-level Searching Process. We utilize the observed *patterns* to reduce the search space of finding the optimal TLP combination. We first ensure that the TLP values are high enough to sufficiently utilize the shared resources. Subsequently, we find the *critical* application and its TLP value that leads to the inflection point in the EB-based metrics. Once the TLP of the *critical* application is fixed, the TLP value of the non-critical application is tuned to further improve the EB-based metric. Because of the existence of the *patterns*, we expect that the *critical* application's TLP still leads to a inflection point regardless of the changes in TLP of the non-critical application. This final stage of tuning is similar to the one application scenario, where tuning of TLP is performed for optimizing the effective bandwidth (Section III-A). We find that this searching process based on the *patterns* (i.e., *pattern-based searching (PBS)*) is an efficient way (reduces the number of TLP combinations to search) to find the appropriate TLP combination targeted for optimizing a specific EB-based metric. In this context, we propose three PBS mechanisms: PBS-WS, PBS-FI, and PBS-HS to optimize for Weighted Speedup (WS), Fairness Index (FI), and Harmonic Weighted Speedup (HS), respectively.

B. Optimizing WS via PBS-WS

As per our discussions in Section IV, our goal is to find the TLP combination that would lead to the highest total effective bandwidth (EB-WS). We describe this searching process for two-application workloads, however, it can be trivially extended for three or more application workloads as described later in Section VI-D.

Consider Figure 6(a) that shows the EB-WS for the workload BLK_TRD. We show individual EB values of each application, that is, EB-BLK and EB-TRD in Figure 6(b). The *pattern* demonstrating sharp drop in EB-WS (i.e., inflection points) is in the shaded region. We follow the high-level searching process described earlier. First, when both applications execute with TLP values of 1, the EB-WS is low (0.55) due to low DRAM bandwidth utilization (29%, not shown). Therefore, as per Guideline-1, this TLP combination is not desirable.

Second, we focus on finding the *critical* application. The process is as follows. We execute each application with TLP of 1, 2, 4, 8 etc. by keeping the TLP of the other application to be fixed at 24. The TLP value of 24 ensures that the GPU system is not under-utilized. This process is repeated for every application in the workload. The application that exhibits a larger drop in EB-WS is *critical* and its TLP is fixed. We decide BLK as the *critical* application as it affects EB-WS the most

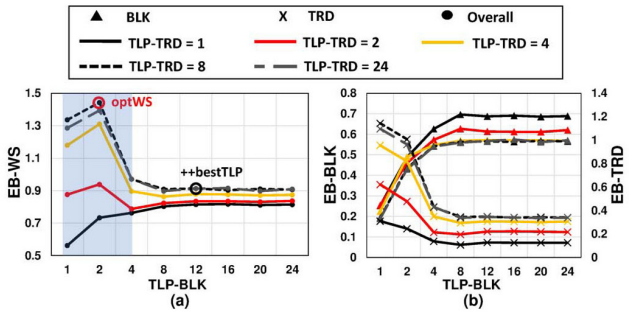


Fig. 6: Illustrating the patterns observed in `BLK_TRD`.

(Figure 6(a)) – the sharp drop in EB-WS after TLP-BLK=2 is prominent.

Third, the next step is to tune the TLP for the non-critical application to reduce the contention and further improve the EB-WS. The searching for TLP of the non-critical application is stopped when the EB-WS no more increases. Therefore, in our example, after fixing TLP-BLK to 2, we start tuning the TLP-TRD to further optimize the EB-WS. The searching process stops at the TLP-TRD = 8, leading to the optimal TLP combination to be (2,8), which is also the optWS[10]. As evident, the whole search process requires only a few samples and does not require an exhaustive search across all combinations.

C. Optimizing Fairness via PBS-FI

In this scheme, our goal is to find a TLP combination that would lead to a better balance between the individual effective bandwidth of the co-scheduled applications. Therefore, we strive to find the TLP combination that would lead to the highest EB-FI. For all the evaluated workloads, we find that a *pattern* also exists in their EB-FI curves (not shown). As a result, we are able to first find the *critical* application that affects the EB-FI the most, followed by tuning the TLP of other non-critical applications.

To intuitively understand the searching process, we study the EB-difference between two applications and plot this difference against TLP to understand the trends in them. A lower absolute value of the difference indicates a fairer system (higher EB-FI) as the EB values of applications are similar (see Section IV).

Consider the example of `BLK_TRD` in the context of fairness. Figure 7 (a) and (b) show two different views of the same data related to EB-difference – one being TLP-BLK as x-axis and curves representing iso-TLP-TRD states (Figure 7 (a)), and vice versa for the second view (Figure 7 (b)). We examine the effect on EB-difference when the TLP of a particular application changes with the TLP of the other application fixed at 24. This process is repeated for every application in the workload. The application that causes larger changes in EB-difference is considered to be *critical*. For example, in Figure 7 (a) and (b), `BLK` is more *critical* than `TRD` because changes in TLP-BLK of `BLK` induces larger changes in the EB-difference, when TLP-TRD is kept constant at 24 (Figure 7 (a)). We then keep the *critical* application's TLP fixed (e.g., TLP-BLK is 2),

[10]There is a possibility that this final process of tuning can free up just enough resources so that the infection point of EB-WS shifts to the right (i.e., *pattern* does not hold), leading to a sub-optimal TLP combination. However, we never observed such a scenario in our experiments.

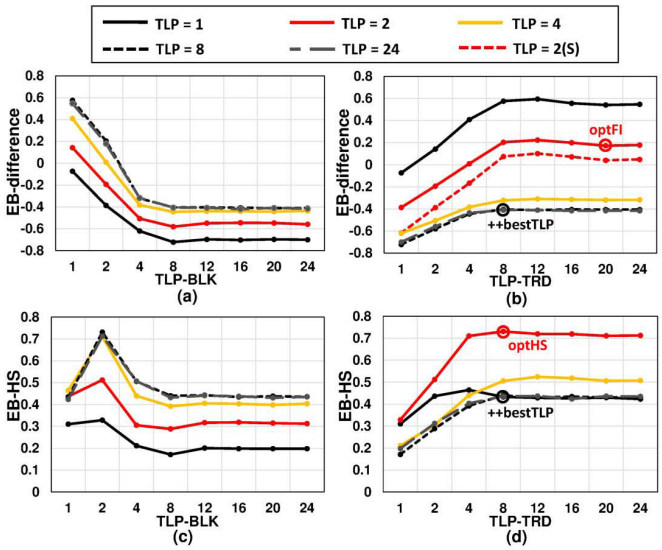


Fig. 7: Illustrating the working of PBS-FI (a & b) and PBS-HS (c & d) schemes for `BLK_TRD`.

where EB-difference is near zero. After fixing, we start to tune the TLP of the other application (i.e., `TRD`). The searching is stopped when the lowest absolute EB-difference is found.

We observe that this searching process stops when TLP-TRD is 4 (Figure 7 (b)). However, optFI is (2,20) instead of (2,4). This difference is caused because the EB-FI uses scaling factor that is approximately calculated by either sampling or user-given group information. We plot the curve (dashed red line, Figure 7 (b)) with the exact scaling factor (Table IV). We are able to locate the correct optFI (2,20) as that point is the closest to 0 on the dashed red line.

D. Optimizing HS via PBS-HS

In this scheme, our goal is to optimize EB-WS. We again take advantage of the *patterns* and observations discussed earlier in the context of PBS-WS and PBS-FI. For all the evaluated workloads, we find that a *pattern* exists in their EB-HS curves. Therefore, we are able to first find the *critical* application that affects the EB-HS the most, followed by TLP tuning of the non-critical application.

Consider the example of `BLK_TRD` in the context of HS. Figure 7 (c) and (d) show two different views of the same data related to EB-HS metric – one being TLP-BLK as x-axis and curves representing iso-TLP-TRD states (Figure 7 (c)), and vice versa for the second view (Figure 7 (d)). PBS-HS starts with examining the effect on EB-HS when the TLP of a particular application changes and the TLP of the other application is fixed at 24. This process is repeated for every application in the workload. The application that causes larger drops in EB-HS value is considered to be *critical*. For example, in Figure 7 (c) and (d), *BLK* is again the *critical* application as it affects the EB-HS the most (larger drop in TLP-TRD=24 curve as TLP-BLK increases, Figure 7(a)). After fixing TLP-BLK to be 2, we start tuning TLP-TRD so as to further optimize the EB-HS. The searching process stops at TLP-TRD=8, leading to the optimal combination of (2,8), which is exactly the optHS.

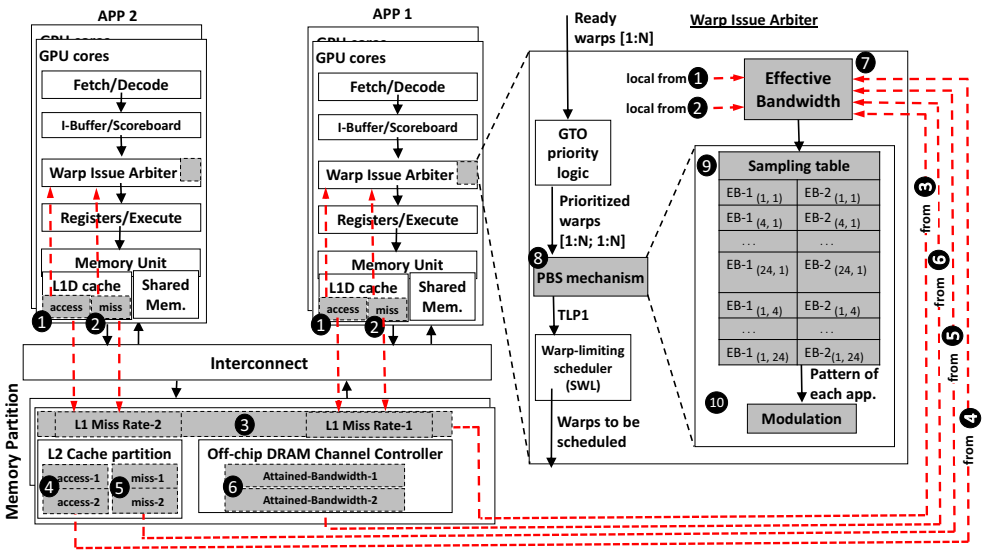


Fig. 8: Proposed hardware organization. Additional hardware is shown via shaded components and dashed arrows.

E. Implementation Details and Overheads

Our mechanism requires periodic sampling of cache miss rates at L1 and L2, and memory bandwidth utilization. In our experiments, we observe uniform miss rate and bandwidth distribution among the memory partitions and uniform L1 miss rates across cores that execute the same application. Therefore, to calculate EB in a low-overhead manner, instead of calculating EB by collecting information from every core and L2/memory partition, we collect: a) L1 miss rate information only from one core per application, and b) attained bandwidth and L2 miss rate information of every application only from one of the L2/memory partitions.

Figure 8 shows the architectural view of our proposal. First, after each sampling period, we use the total number of L1 data cache accesses (❶) and misses (❷), from each designated core, and calculate the miss rate of the application. The calculated miss rates are sent through the interconnect to the designated memory partition and stored in their respective buffers (❸). Then, the miss rate of each application, L2 cache accesses (❹) and misses (❺), and attained bandwidth (❻) from the designated memory partition are forwarded to each core to be used along with the locally collected L1 data. Such data is used, per core, to calculate EB (❼). The calculated EB values are then fed to our PBS mechanism (❽), which resides inside the warp issue arbiter within each core, and stored in a small table (❾). In this table, each line represents the EB of both applications, corresponding to the TLP combination used for the current sampling period (indicated by the subscript). After sampling, our PBS mechanism extracts the pattern from each application and then changes the TLP value for one application accordingly. The next step, modulation (❿), varies the TLP value of the other application to maximize the relevant EB-based metric. Finally, the calculated TLP is sent to the warp-limiting scheduler.

We break down overhead in terms of storage, computation, and communication. In terms of storage, two 12-bit registers per core, and three 12-bit registers and one 15-bit register per memory partition are required to track per-application L1 miss rate, L2 miss rate, and BW, respectively. The sampling table needs 60 bytes. In terms of computation, the sampled data is fed to the PBS mechanism module (❽), which performs a simple search over the 16 $\times$ 2 samples collected over the sampling window. In terms of communication, using a crossbar, the designated memory partition relays the collected information (12 bits $\times$ 6 + 15 bits $\times$ 2 = 102 bits) to the cores every sampling window. We conservatively assume that the counter values are sent to the cores with a latency of 20 cycles.

All the runtime overheads are modeled in the PBS results presented in Section VI. We empirically find that a monitoring interval of 3000 cycles for each TLP combination searched via PBS is sufficient as trends do not change significantly beyond 3000 cycles. The PBS is re-started when any kernel is re-launched.

VI. Experimental Evaluation

In this section, we evaluate our proposed PBS schemes (Section V) and compare them against ++bestTLP, opt (optWS, optFI, optHS), and the following additional schemes:

Brute-Force (BF). BF scheme performs an offline exhaustive search across all the possible TLP combinations (64) to find the one that provides the best EB-based metric. Therefore, BF has three different versions: BF-WS, BF-FI, and BF-HS that optimizes EB-WS, EB-FI, and EB-HS, respectively. BF schemes provide a good estimate of the potential of improving the SD-based metrics (the ones we finally report) via optimizing EB-based runtime metrics. Note that opt schemes are also brute-force but instead they perform an exhaustive search to find the best SD-based metric.

PBS (Offline). PBS-Offline schemes follow the exact same procedure as previously described in the PBS schemes (Section V) but do not consider: a) any runtime overheads, and b) dynamic changes in interference across different kernel executions in the workload. We consider this comparison point to decouple the runtime effects from the inherent benefits of the proposed

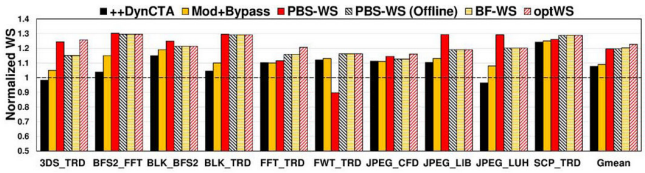


Fig. 9: Impact of our schemes on Weighted Speedup. Results are normalized to ++bestTLP.

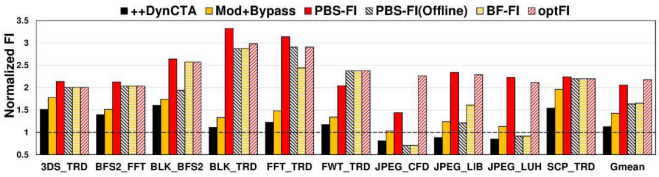
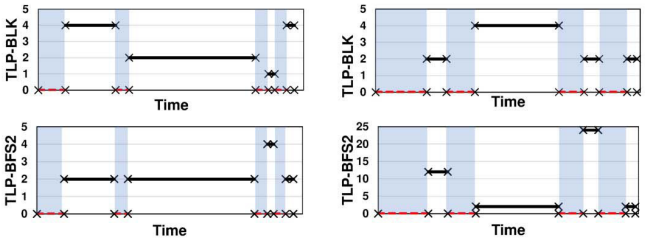


Fig. 10: Impact of our schemes on Fairness. Results are normalized to ++bestTLP.



(a) Optimizing WS **(b) Optimizing FI**

Fig. 11: Effect of changes in TLP over time for BLK_BFS2 with: a) PBS-WS and b) PBS-FI.

schemes. Similar to PBS, PBS-Offline also has three versions: PBS-WS (Offline), PBS-FI (Offline), and PBS-HS (Offline).

Mod+Bypass [35]. In addition to ++DynCTA, we compare the PBS mechanisms against the recently proposed TLP management mechanism Mod+Bypass [35] for multi-application scenario. They use both CTA modulation and cache bypassing mechanism to enhance the system throughput.

A. Effect on Weighted Speedup

Figure 9 shows the impact of different schemes on the WS for 10 representative workloads (out of 50 evaluated) along with Gmean across evaluated workloads. The results are normalized to the WS obtained under ++bestTLP. Six observations are in order. First, on average, benefits of BF-WS are as good as optWS (within 2%) implying that optimizing EB-WS is a good candidate to improve SD-based WS. However, as EB-WS is a proxy for WS and it may not work for all workloads as discussed in Section IV. As the number of outliers are very few, we did not consider scaling factors for optimizing WS. Second, PBS-WS (Offline) overall performs as good as optWS. This shows the inherent effectiveness of PBS in finding the TLP combination that results in higher WS over ++bestTLP.

Third, on average, PBS also performs as good as the PBS-WS (Offline). Note that PBS-WS (Offline) offers a trade-off. As it is a static technique it does not incur runtime overhead but also cannot adapt to different runtime interference patterns for locating better TLP combination within the same workload execution. Therefore, we observe that the benefits of PBS-WS can be: 1) similar to PBS-WS (Offline) (e.g., `BLK_TRD`), where the runtime benefits cancel out with the overheads; 2) worse than PBS-WS (Offline) (e.g., `FWT_TRD`, where the runtime overheads hamper the WS; or 3) better than PBS-WS (Offline) (e.g., `3DS_TRD`, `BLK_BFS2`), where the runtime tuning of TLP combination provides benefits. To illustrate the last point, Figure 11(a) shows the dynamic changes in TLP (TLP-BLK above and TLP-BFS2 below) over the course of `BLK_BFS2` execution. The shaded areas represent the sampling period including the time during which the decision cannot be taken because the execution time of the kernel is too short. As expected and discussed before in Section III, (2,2) is the most preferred TLP combination for `BLK_BFS2` and is chosen for the longest duration of time. Other TLP combinations are chosen during other time intervals to boost the WS further.

Fourth, ++DynCTA provides additional benefits over ++bestTLP (7% on average) because of its ability to adapt under a shared environment. However, it is still far from PBS and other schemes as ++DynCTA attempts to enhance the performance based on application's local information and hence can overwhelm the memory system. Fifth, Mod+Bypass technique helps in improving the performance further over ++DynCTA mainly because it also bypasses the application that does not take advantage of caches, thereby reducing the cache contention. However, this mechanism is still far from optWS as it does not consider the memory bandwidth consumption and the combined effects of TLP modulation. Finally, PBS-WS performs significantly better (20%, on average) than the ++bestTLP because of the reasons extensively discussed in Section III and Section V. `FWT_TRD` is the only exception as PBS-WS is not able to find the optimal TLP combination due to a smaller sampling period.

B. Effect on Fairness Index

Figure 10 shows the impact of different schemes on FI for 10 representative workloads (out of 50 evaluated) along with Gmean across evaluated workloads. The results are normalized to the FI obtained under ++bestTLP. Five observations are in order. First, on average, benefits of BF-FI are not as close as optFI implying that runtime optimizations play an important role in achieving high fairness. To evaluate the impact of scaling factor, we calculated BF-FI both using grouping as well as sampling information (Section IV). We find that BF-FI calculated using grouping information is 16% (not shown) better in FI, averaged across all workloads. However, the grouping information needs to be supplied by the user. If exact scaling factors are used (Table IV), BF-FI is close to optFI as expected. For a fair comparison, Figure 10 shows the sampling-based BF-FI for comparisons against other dynamic counterparts.

Second, PBS-FI (Offline) overall performs as good as BF-FI implying that the scheme itself is effective in providing high fairness. Third, PBS-FI is able to capture the runtime effects well and is better than PBS-FI (Offline) in many workloads. As an example, Figure 11(b) shows the dynamic changes in TLP (TLP-BLK above and TLP-BFS2 below) over the course of `BLK_BFS2` execution. The shaded areas represent the sampling period. We observe that despite higher sampling

overhead (as it includes additional sampling to calculate the scaling factor), PBS-FI is able to provide much higher benefits than other schemes. It is because TLP combination of (4,2) allowed to reduce the slowdown of BLK while preserving the slowdown for BFS2 when it was executing non-cache sensitive kernels. Fourth, ++DynCTA and Mod+Bypass provide additional benefits over ++bestTLP (12% and 42% on average, respectively) because of their ability to adapt under a shared environment. However, both ++DynCTA and Mod+Bypass themselves are not designed to improve fairness in the multi-application environment and only focus on performance. Finally, PBS-FI performs significantly better (2×, on average) than the ++bestTLP because of the reasons discussed before.

C. Effect on Harmonic Weighted Speedup

Figure 12 shows the impact of the schemes on all the evaluated 50 workloads. For brevity, we do not show the results for each workload separately. Instead, we use the grouping information (Table IV) to form 2-application clusters and report the average (geometric mean) of the HS improvements of every workload in the cluster. As there are four groups, 10 such clusters are possible. Figure 12 shows the results of nine such clusters. We do not study G11 cluster as both the applications belonging to that cluster have low individual EB and interference. We observe that PBS-HS enhances HS on average by 15% over ++bestTLP. Additionally, compared to optHS, PBS-HS performance is behind by 2%, on average. We conclude that PBS-HS is a technique that can significantly enhance both system throughput and fairness over ++bestTLP.

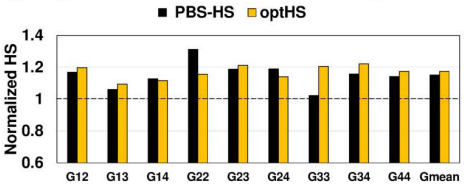


Fig. 12: Impact of our schemes on HS.

D. Case Studies

We perform four sensitivity studies to understand the impact of the proposed schemes under different core and cache partitioning, application scaling, and memory scheduling scenarios.

Core partitioning. We test PBS under two different core partitioning scenarios: 2:1 and 1:2 partitioning (e.g., in 2:1, the first and the second applications are assigned 20 and 10 cores, respectively) and compare it to our baseline that allocates 15 cores to each application. Figure 13(a) shows the benefits of PBS over ++bestTLP observed in WS, FI, and HS averaged across all our 50 workloads. The bar denoted by Equal represents the average improvement of PBS with equal partitioning, and the bar denoted by Unequal represents the average improvement of PBS with the best performing partitioning scheme among 2:1 and 1:2 (found separately for each workload and then averaged). PBS enhances WS, FI, and HS under both equal and unequal core partitioning scenarios, compared to ++bestTLP. However, the benefits of PBS reduce with unequal partitioning because with different core partitioning each application executes with a different amount of TLP. As we choose the best (among 2:1 and 1:2) core partitioning configuration, the interference is also alleviated because of core partitioning itself in addition to our TLP management schemes. As the design of core partitioning is itself an interesting and non-trivial research problem, we conclude that these techniques still do not completely solve the interference problem and TLP management techniques like PBS can provide additional benefits.

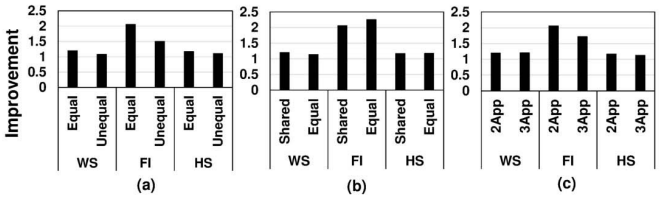


Fig. 13: Effect of PBS over ++bestTLP with: a) core partitioning, b) cache partitioning, c) 3-application scaling.

Cache partitioning. Figure 13(b) shows the benefits of PBS when the L2 is way-partitioned equally (denoted by Equal) across two applications, and compare it to our baseline where the L2 cache is shared (denoted by Shared). We observe that PBS improves all three metrics (WS, FI, and HS) under both the scenarios. Cache partitioning can alleviate the interference at L2, but it might be suboptimal in scenarios where different applications in the same workload utilize the caches differently. This might lead to a portion of the cache to be not utilized well. On the other hand, PBS changes the cache demand of each application by TLP modulation.

Application Scalability. For a k-application workload, we first rank the criticality of each application in the workload based on the magnitude of the EB drop. While determining the ranking, the TLP of other applications is fixed at 24 (same as discussed before in Section V). If N is the number of TLP choices, the procedure for deciding the criticality takes less than $N \times k$ steps. Subsequently, the tuning of TLP of each application would take N $\times$ $(k-1)$ steps. Therefore, the associated overall search complexity is linear to the number of applications (O(N $\times$ k)). In this case study, we evaluate PBS on three-application workloads by performing some straightforward extensions to the PBS. Therefore, we find the critical application one by one under the interference of the other two remaining applications, keeping other steps the same. We choose 20 representative three-application workloads to compare the average benefits to that of workloads with two applications (Figure 13(c)). We observe that the benefits of PBS are reasonably stable as the number of applications scales. We conclude that our techniques are not limited to workloads that consist of only two applications.

Memory scheduling. We find that PBS provides significant benefits (15% in WS and 1.92× in FI, on average across 50 workloads) over WEIS memory scheduler designed for multi-GPU execution [26]. We conclude that TLP management techniques are more effective than the previously proposed memory scheduling techniques.

VII. Related Work

To our knowledge, this is the first work that proposes TLP management techniques for improving system throughput and

fairness in a multi-application environment for GPUs. In this section, we outline some particularly relevant works.

TLP management techniques in GPUs. Rogers *et al.* proposed a mechanism that limits the TLP based on the level of thrashing in each core's private L1 data cache [50]. Kayiran *et al.* proposed a TLP optimization technique that works based on the latency tolerance of individual GPU cores [30]. Jia *et al.* proposed a mechanism that consists of a reference reordering technique and a bypassing technique such that the cache thrashing and also resource stalls at the caches reduce [23]. The input to this mechanism is from the L1 caches, and the decision is local. Sethia and Mahlke devised a method that controls the number of threads and core/memory frequency of GPUs [54]. Sethia *et al.* used a priority mechanism that allows better overlapping of computation and memory accesses by limiting the number of warps that can simultaneously access memory [53]. The work by Zheng *et al.* allows the execution of many warps concurrently without thrashing the L1 cache by employing cache bypassing [69]. All these works use *local* metrics available at the GPU cores and do not consider resource contention at the L2 caches and the memory. However, we propose mechanisms where *all GPU applications* control their TLP while being *aware of each other*. Further, our mechanisms are optimized for improving system throughput and fairness and *not* instruction throughput, which was the focus of aforementioned works.

Hong *et al.* proposed various analytical methods for estimating the effect of TLP on performance [20], [21], but do not propose run-time mechanisms. Kayiran *et al.* devised a mechanism where GPU cores modulate their TLP based on system-level congestion when GPU applications execute alongside CPU applications in a shared environment [32]. Their mechanism uses system-level metrics to unilaterally control the TLP of a *single GPU application*, whereas our mechanisms control TLP while being aware of *all applications* in the system.

Concurrent execution of multiple applications on GPUs. Xu *et al.* proposed running multiple GPU applications *on the same GPU cores* and assigning CTAs slots to different applications to improve resource utilization of GPU cores [67]. Likewise, Wang *et al.* proposed running multiple GPU applications *on the same GPU cores*, and augmented it with a warp scheduler that adopts a time-division multiplexing mechanism for the co-running applications based on *static profiling* [66]. These intra-core partitioning techniques are used to partition resources within a core. However, co-running kernels interfere with each other significantly, especially in small L1 GPU caches. In such cases, running these kernels separately on different cores can be more effective for avoiding intra-core contention. GPU Maestro dynamically chooses between intra-core and inter-core techniques to reap the benefits of both [45]. However, none of these mechanisms change the shared cache and memory footprint of each application, and thus directly alleviate the shared memory interference. Our goal is to address the shared resource contention in L2 caches and main memory by managing TLP of each application differently. PBS allows each application to dynamically change its cache and memory footprint cognizant of the other applications' state. Pai *et al.* proposed elastic kernels that allow a fine-grained control over their resource usage [43]. Their work targets increasing the utilization of computing resources by accounting for the parallelism limitation imposed by the hardware, whereas our mechanism considers the memory system contention to modulate parallelism. Li *et al.* proposed a technique to adjust TLP of concurrently executing kernels [35], which we quantitatively and qualitatively compare in Section VI. We conclude that even if a new resource partitioning technique (see case study (Section VI-D)) is employed, the problem of multi-application contention in the memory system remains.

Cache and memory management. In the context of traditional CPUs, several works have investigated coordinated cache and memory management, and throttling for lower memory system contention. Zahedi *et al.* proposed a game-theory based approach for partitioning cache capacity and memory bandwidth for multiple software agents [68]. Ebrahimi *et al.* proposed a throttling technique that improves system fairness and performance in multi-core memory systems [10]. Eyerman *et al.* analyzed the effects of varying degrees of TLP on performance, in various multi-core designs [12]. Heirman *et al.* proposed a technique that matches the application's cache working set size and off-chip bandwidth demand with the available system resources [19]. Qureshi and Patt proposed a cache partitioning mechanism in the context of multi-core CPUs [48]. Their mechanism, based on the cache demand of each application, allocates cache space to co-running applications, whereas our mechanism changes the cache demand of each application by controlling their TLP.

VIII. Conclusions

This paper analyzed the problem of shared resource contention between multiple concurrently executing GPGPU applications and showed that there is an ample scope for TLP management techniques for improving system throughput and fairness in GPUs. Our detailed analysis showed that these metrics are highly correlated with *effective bandwidth*, which is defined as the ratio of attained DRAM bandwidth to the combined cache miss rate. Based on this observation, we designed pattern-based effective bandwidth management schemes to quickly locate the most efficient and fair TLP configuration for each application. Results show that our proposed techniques can significantly improve the system throughput and fairness in GPUs compared to previously proposed state-of-the-art mechanisms. While this paper focused on a specific platform with concurrently executing GPU applications, we believe that the presented analysis and the insights can be extended to other systems (e.g., chip-multiprocessors, systems-on-chip with accelerator IPs, server processors) where contention in shared caches and memory resources are performance-critical factors.

Acknowledgment

The authors would like to thank the anonymous reviewers and the members of Insight Computer Architecture Lab at the College of William and Mary for their feedback. This material is based upon work supported by the National Science Foundation (NSF) grants (#1657336 and #1717532) and a start-up grant from the College of William and Mary. This work was performed in part using computing facilities at the College of William and Mary which were provided by contributions from the NSF, the Commonwealth of Virginia Equipment Trust Fund

and the Office of Naval Research. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] "NVIDIA GTX 780-Ti," http://www.nvidia.com/gtx-700-graphics-cards/gtx-780ti/.

[2] J. Adriaens *et al.*, "The Case for GPGPU Spatial Multitasking," in *HPCA*, 2012.

[3] Advanced Micro Devices Inc., "AMD Graphics Cores Next (GCN) Architecture," Advanced Micro Devices, 2012.

[4] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[5] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[6] A. Danalis *et al.*, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *GPGPU*, 2010.

[7] R. Das *et al.*, "Application-to-core Mapping Policies to Reduce Memory Interference in Multi-core Systems," in *PACT*, 2012.

[8] R. Das *et al.*, "Application-aware Prioritization Mechanisms for on-chip Networks," in *MICRO*, 2009.

[9] R. Das *et al.*, "Aergia: Exploiting Packet Latency Slack in on-chip Networks," in *ISCA*, 2010.

[10] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *ASPLOS*, 2010.

[11] A. Eklund *et al.*, "Medical Image Processing on the GPU-Past, Present and Future," *Medical Image Analysis*, 2013.

[12] S. Eyerman and L. Eeckhout, "The Benefit of SMT in the Multi-core Era: Flexibility Towards Degrees of Thread-level Parallelism," in *ASPLOS*, 2014.

[13] X. Gong *et al.*, "TwinKernels: An Execution Model to Improve GPU Hardware Scheduling at Compile Time," in *CGO*, 2017.

[14] N. Goswami *et al.*, "Power-performance Co-optimization of Throughput Core Architecture Using Resistive Memory," in *HPCA*, 2013.

[15] GPGPU-Sim v3.2.1. Address mapping. Available: {http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual#Memory_Partition}

[16] GPGPU-Sim v3.2.1. GTX 480 Configuration. Available: {https://dev.ece.ubc.ca/projects/gpgpu-sim/browser/v3.x/configs/GTX480}

[17] C. Gregg *et al.*, "Fine-grained Resource Sharing for Concurrent GPGPU Kernels," in *HotPar*, 2012.

[18] Z. Guz *et al.*, "Many-Core vs. Many-Thread Machines: Stay Away from the Valley," *CAL*, January 2009.

[19] W. Heirman *et al.*, "Undersubscribed Threading on Clustered Cache Architectures," in *HPCA*, 2014.

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

[21] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *ISCA*, 2010.

[22] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0. Available: {http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf}

[23] W. Jia *et al.*, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *HPCA*, 2014.

[24] A. Jog *et al.*, "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *GPGPU*, 2014.

[25] A. Jog *et al.*, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.

[26] A. Jog *et al.*, "MAFIA - Multiple Application Framework in GPU Architectures," *URL: https://github.com/adwaitjog/mafia*, 2015.

[27] A. Jog *et al.*, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[28] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[29] I. Karlin *et al.*, "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application," in *IPDPS*, 2013.

[30] O. Kayiran *et al.*, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.

[31] O. Kayiran *et al.*, "μC-States: Fine-grained GPU Datapath Power Management," in *PACT*, 2016.

[32] O. Kayiran *et al.*, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.

[33] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[34] J. Lee *et al.*, "Orchestrating Multiple Data-Parallel Kernels on Multiple Devices," in *PACT*, 2015.

[35] X. Li and Y. Liang, "Efficient Kernel Management on GPUs," in *DATE*, 2016.

[36] J. Liu *et al.*, "SAWS: Synchronization Aware GPGPU Warp Scheduling for Multiple Independent Warp Schedulers," in *MICRO*, 2015.

[37] K. Menychtas *et al.*, "Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators," in *ASPLOS*, 2014.

[38] NVIDIA, "How to Harness Big Data for Improving Public Health," http://www.govhealthit.com/news/how-harness-big-data-improving-public-health.

[39] NVIDIA, "Researchers Deploy GPUs to Build World's Largest Artificial Neural Network," http://nvidianews.nvidia.com/Releases/Researchers-Deploy-GPUs-to-Build-World-s-Largest-Artificial-Neural-Network-9c7.aspx.

[40] NVIDIA, "CUDA C/C++ SDK Code Samples," http://developer.nvidia.com/cuda-cc-sdk-code-samples, 2011.

[41] NVIDIA, "JP Morgan Speeds Risk Calculations with NVIDIA GPUs," 2011.

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

[43] S. Pai *et al.*, "Improving GPGPU Concurrency with Elastic Kernels," in *ASPLOS*, 2013.

[44] J. J. K. Park *et al.*, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *ASPLOS*, 2015.

[45] J. J. K. Park *et al.*, "Dynamic Resource Management for Efficient Utilization of Multitasking GPUs," in *ASPLOS*, 2017.

[46] S. I. Park *et al.*, "Low-cost, High-speed Computer Vision Using NVIDIA's CUDA Architecture," in *AIPR*, 2008.

[47] M. K. Qureshi *et al.*, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.

[48] M. K. Qureshi and Y. N. Patt, "Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006.

[49] M. Rhu *et al.*, "A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures," in *MICRO*, 2013.

[50] T. G. Rogers *et al.*, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[51] D. Schaa and D. Kaeli, "Exploring the Multiple-GPU Design Space," in *IPDPS*, 2009.

[52] I. Schmerken, "Wall Street Accelerates Options Analysis with GPU Technology," 2009.

[53] A. Sethia *et al.*, "Mascar: Speeding up GPU Warps by Reducing Memory Pitstops," in *HPCA*, 2015.

[54] A. Sethia and S. Mahlke, "Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution," in *MICRO*, 2014.

[55] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.

[56] S. S. Stone *et al.*, "Accelerating Advanced MRI Reconstructions on GPUs," *JPDC*, vol. 68, no. 10, pp. 1307–1318, 2008.

[57] J. A. Stratton *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois, at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[58] I. Tanasic *et al.*, "Enabling Preemptive Multiprogramming on GPUs," in *ISCA*, 2014.

[59] X. Tang *et al.*, "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *HPCA*, 2017.

[60] Y. Ukidave *et al.*, "Runtime Support for Adaptive Spatial Partitioning and Inter-kernel Communication on GPUs," in *SBAC-PAD*, 2014.

[61] Y. Ukidave *et al.*, "Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning," in *IPDPS*, 2016.

[62] T. Vijayaraghavany *et al.*, "Design and Analysis of an APU for Exascale Computing," in *HPCA*, 2017.

[63] N. Vijaykumar *et al.*, "Enabling Efficient Data Compression in GPUs," in *ISCA*, 2015.

[64] J. Wang *et al.*, "Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs," in *ISCA*, 2015.

[65] L. Wang *et al.*, "Exploiting Concurrent Kernel Execution on Graphic Processing Units," in *HPCS*, 2011.

[66] Z. Wang *et al.*, "Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-grained Sharing," in *HPCA*, 2016.

[67] Q. Xu *et al.*, "Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming," in *ISCA*, 2016.

[68] S. M. Zahedi and B. C. Lee, "REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors," in *ASPLOS*, 2014.

[69] Z. Zheng *et al.*, "Adaptive Cache and Concurrency Allocation on GPGPUs," *CAL*, vol. 14, no. 2, pp. 90–93, 2015.