Coordinated CTA Combination and Bandwidth Partitioning for GPU Concurrent Kernel Execution

ZHEN LIN, North Carolina State University, USA
HONGWEN DAI, North Carolina State University, USA
MICHAEL MANTOR, Advanced Micro Devices, USA
HUIYANG ZHOU, North Carolina State University, USA

Contemporary GPUs support multiple kernels to run concurrently on the same streaming multiprocessors (SMs). Recent studies have demonstrated that such concurrent kernel execution (CKE) improves both resource utilization and computational throughput. Most of the prior works focus on partitioning the GPU resources at the cooperative thread array (CTA) level or the warp scheduler level to improve CKE. However, significant performance slowdown and unfairness are observed when latency-sensitive kernels co-run with bandwidth-intensive ones. The reason is that bandwidth over-subscription from bandwidth-intensive kernels leads to much aggravated memory access latency, which is highly detrimental to latency-sensitive kernels. Even among bandwidth-intensive kernels, more intensive kernels may unfairly consume much higher bandwidth than less intensive ones.

In this paper, we first make a case that such problems cannot be sufficiently solved by managing CTA combinations alone and reveal the fundamental reasons. Then, we propose a coordinated approach for CTA combination and bandwidth partitioning. Our approach dynamically detects co-running kernels as latency sensitive or bandwidth intensive. As both the DRAM bandwidth and L2-to-L1 Network-on-Chip (NoC) bandwidth can be the critical resource, our approach partitions both bandwidth resources coordinately along with selecting proper CTA combinations. The key objective is to allocate more CTA resources for latency-sensitive kernels and more NoC/DRAM bandwidth resources to NoC-/DRAM-intensive kernels. We achieve it using a variation of dominant resource fairness (DRF). Compared with two state-of-the-art CKE optimization schemes, SMK [52] and WS [55], our approach improves the average harmonic speedup by 78% and 39%, respectively. Even compared to the best possible CTA combinations, which are obtained from an exhaustive search among all possible CTA combinations, our approach improves the harmonic speedup by up to 51% and 11% on average.

CCS Concepts: • **Computer systems organization** → *Single instruction, multiple data*;

Additional Key Words and Phrases: GPGPU, TLP, bandwidth management, concurrent kernel execution

1 INTRODUCTION

To deliver high throughput, GPUs incorporate a large amount of computational resources and support high memory bandwidth. However, the resource demands across different GPU kernels vary significantly, which may lead to saturation of certain resources and underutilization in others. One solution to such unbalanced resource utilization is to concurrently execute multiple kernels with complementary characteristics. Prior works [11, 42, 43, 52, 55] have shown that both the GPU utilization and throughput can be improved by co-running heterogeneous kernels. Moreover, with GPUs being increasingly deployed in cloud servers, there is a strong need for GPU resource to be shared among multiple users.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
XXXX-XXXX/2019/1-ART1
https://doi.org/10.1145/3326124

There are different ways to support concurrent kernel execution (CKE) on GPUs. A simple way is to assign different kernels to different sets of streaming multiprocessors (SMs), such as Spatial Multitasking [1]. However, as pointed out in prior works [11, 43, 52, 55], Spatial Multitasking does not address resource underutilization within SMs. SMK [52], WS, [55] and Maestro [43] are the recent works focusing on intra-SM sharing. They propose different Cooperative Thread Array (CTA) combination algorithms to determine how many CTAs from each kernel should be dispatched to the same SMs. To achieve fair partitioning, SMK leverages a 'Dominant Resource Fairness' metric to fairly partition the static resources (such as the register file and shared memory) to each kernel. To further ensure fair performance, SMK periodically assigns each kernel a fixed time quota in the warp scheduler. The time quota partitioning is based on profiling of the standalone execution of each individual kernel. In comparison, WS first determines the scalability curves, i.e., the performance vs. the number of CTAs per SM, for each individual kernels. Then WS uses the scalability curves to determine the CTA combination that generates the minimum combined performance slowdown. Maestro proposes a dynamic search approach to find the optimal CTA combination. In addition, Maestro argues for loose round-robin (LRR) as the kernel-aware scheduling policy for CKE.

In this paper, we highlight that memory interference can significantly affect the throughput and fairness of CKE. And we make a case that even the optimal CTA combination does not eliminate the negative memory interference impact. To address this problem effectively, we propose a coordinated approach for CTA combination and bandwidth partitioning. Our proposed approach is based on the following observations.

First, bandwidth over-subscription from the bandwidth-intensive kernels introduce high queuing delay and drastically increase the memory latency, which may lead to severe performance degradation for other co-running kernels. Although it is well-known that compute-intensive kernels leverage massive thread(warp)-level parallelism to hide the memory latency, the increased memory latency is so high (up to 8.2x higher than the standalone execution) that it can hardly be fully hidden. In this paper, we refer to the kernels which are sensitive to the increased memory latency as latency sensitive.

Second, we observe that, besides the DRAM bandwidth, the L2-to-L1 NoC bandwidth can be a critical resource in the GPU memory system. The reason is that all the data replied either from the L2 cache or DRAM go through the L2-to-L1 NoC. As a result, the L2-to-L1 NoC can become a bottleneck when the L2 cache filters a substantial amount of memory requests and the DRAM bandwidth is not saturated. Therefore, both the NoC bandwidth and the DRAM bandwidth need to be managed carefully. In prior works, both DRAM-intensive and NoC-intensive kernels are both considered memory intensive. In this work, we highlight the difference between them and show that these two types of kernels can benefit from CKE, i.e., running them together would improve resource utilization, as they stress different parts of the memory system.

Third, we find that the above-mentioned issues cannot be addressed even with the oracle CTA combination, which is the best CTA combination from an exhaustive search. There are three fundamental reasons: (a) the bursty memory traffic makes CTA-level control less effective; (b) the lack of application-aware memory scheduling fails to prioritize requests from latency-sensitive benchmarks to reduce their memory access latency; and (c) CTA-level management can be too coarse-grained in controlling thread-level parallelism (TLP).

Forth, different kernels favor different GPU resources. The latency-sensitive kernels require high levels of TLP to better hide the memory latency. On the other hand, the DRAM-intensive kernels are more sensitive to the bandwidth utilization and reducing the CTA number from a bandwidth-intensive kernel may not degrade its performance.

In our proposed coordinated approach for CTA combination and bandwidth partitioning (CCBP), we first dynamically detect the kernels as latency sensitive or bandwidth intensive. Among

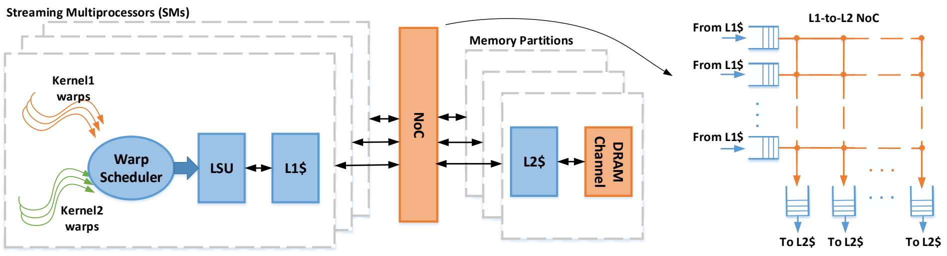


Fig. 1. GPU architecture.

bandwidth-intensive benchmarks, CCBP further classifies them as NoC intensive and DRAM intensive. Then it effectively allocates the CTA number and bandwidth resources for each co-running kernel based on their resource requirements. We derive the bandwidth consumption of both NoC and DRAM as functions of the memory request issue rate and achieve bandwidth allocation by controlling the memory request issue rate according to a kernel's assigned bandwidth quota.

We compare CCBP with three state-of-the-art CKE approaches and the oracle CTA combination, which is the result of an exhaustive search of all possible CTA combinations. Compared to SMK and WS, our approach improves the average harmonic speedup by 78% and 39%, respectively, for 2-kernel co-runs. Compared to a recently proposed approach [11] combining WS and memory-instruction scheduling, our approach improves the average harmonic speedup by 19%. Even compare to the oracle CTA combinations, our approach improves the harmonic speedup by up to 51% and 11% on average.

In this paper, we make the following contributions:

- This work reveals the memory interference problem in GPU CKE cannot be sufficiently solved using CTA-level management alone.
- We highlight that NoC and DRAM bandwidth are different bottlenecks in the GPU memory subsystem. This is the first work that shows it is beneficial to co-run NoC-intensive kernels with DRAM-intensive kernels. To our knowledge, this is also the first work that unifies the management of both the NoC and DRAM bandwidth.
- We propose a dynamic approach to classify GPU kernels into three categories, latency sensitive, NoC intensive or DRAM intensive. Based on the detected kernel types, our scheme selectively assigns resources to the kernels which would benefit the most from such resources.
- Our approach effectively reduces the memory latency for the latency-sensitive kernels. In the meanwhile, the bandwidth utilization is also improved for the bandwidth-intensive kernels.

2 BACKGROUND ON GPU ARCHITECTURE

Contemporary GPU architecture, as illustrated in Figure 1, consists of multiple stream multiprocessors (SMs). Each SM has a private L1 D-cache and all SMs share a multi-banked L2 cache. The L1 caches and L2 cache banks communicate through an NoC (Network-on-Chip).

Each SM hosts a number of cooperative thread arrays (CTAs). Each CTA has a number of warps, each of which is a collection of threads running in the single-instruction multiple-data (SIMD) manner. Every cycle, a warp scheduler selects a ready instruction to issue to the execution pipeline.

In our GPU model, we use two sets of crossbars, one for each direction, for the NoC [3]. The NoC in the L1-to-L2 direction is shown in Figure 1. Our crossbar is a non-blocking interconnect network, which means a connection can be established if both the input and output are idle. If more than one packet needs to reach the same output at the same time, only one packet can be

Table 1. Baseline architecture configuration

Overall config.	16 SMs, 32 threads/warp, 16 L2 banks, 16 DRAM chips
SM config.	1800MHz, 4 schedulers, LRR/GTO warp scheduler
Static resource per SM	32 CTAs, 2048 threads, 256KB register file, 96KB shared memory
L1 D-cache	24KB, 128B block, 8-way associativity, 128 transaction queue entries, 16 store buffer entries, 256 MSHRs
NoC	Two 16*16 crossbars, 32B flit size, bandwidth = 16 ports * 1.2GHz * 32B = 614GB/s
L2 cache per bank	128KB, 128B block, 8-way associativity, 256 MSHRs, WBWA policy
DRAM	1200MHz, 319GB/s bandwidth, FR-FCFS policy, 128 transaction queue entries
Latency	L2 cache latency: 200 cycles, DRAM latency: 380 cycles [37]

Table 2. Benchmark specification

Bench.	Source	Thro.	NoC BW	DRAM BW	L2 Miss	Type
CFD	cfd [8]	7%	54%	14%	15%	NBI
LAV	lavaMD [8]	5%	53%	1%	1%	NBI
SMV	spmv [46]	10%	52%	25%	30%	NBI
KM	kmeans [8]	2%	52%	22%	17%	NBI
LBM	lbm [46]	17%	15%	61%	88%	DBI
FTD	FDTD-2D [17]	29%	32%	65%	66%	DBI
SAD	sad [46]	5%	17%	64%	78%	DBI
SRT	bucketsort [8]	25%	37%	59%	69%	DBI
BFS	bfs [8]	44%	14%	24%	100%	LS
BP	backprop [8]	68%	19%	36%	51%	LS
BT	b+tree [8]	24%	27%	19%	44%	LS
HG	histogram [8]	28%	19%	25%	74%	LS
PF	pathfinder [8]	72%	28%	39%	90%	LS

accepted by the network and other packets have to be queued at their inputs. The L1 miss requests and write requests from multiple SMs share this crossbar to reach their destination L2 banks. When multiple kernels co-run on an SM, they share the L1 cache, the crossbar, and the L2 banks.

Since the Hyper-Q [41] technology was introduced by NVIDA on the Kepler GPUs, recent GPUs support concurrent kernel execution on the same GPUs. The state-of-the-art GPU partitioning schemes [43, 52, 55] leverage intra-SM sharing, which allow warps/CTAs from different kernels to reside on the same SMs. As shown in Figure 1, warps from different kernels contend for the warp scheduler as well as the memory subsystem.

3 METHODOLOGY

3.1 Simulation Specifications

We use GPGPU-sim v3.2.2 to investigate GPU memory interference and evaluate our proposed approach for CKE. The baseline architecture configuration is shown in Table 1. In our experiments, the loose round-robin (LRR) scheduling policy is used by default for CKE and the greedy-then-oldest (GTO) policy is used for standalone kernel running. The memory access latency is configured according to the memory hierarchy study by Mei et al. [37].

3.2 Benchmark Categorization

We studied a wide range of 42 GPU kernels from the Rodinia [8], Parboil [46] and Polybench [17] benchmark suites. Based on their NoC and DRAM bandwidth utilization, the benchmarks are grouped into three categories: NoC intensive (labeled as 'NBI'), DRAM intensive (labeled as 'DBI') and latency sensitive (labeled as 'LS'). To evaluate the mechanisms proposed in this paper, as shown in Table 2, we selected a total of 13 benchmarks using the following criteria. For the NoC-intensive kernels, we ranked the NoC bandwidth utilization of all the 42 kernels and selected the highest 4 kernels. Similarly, for the DRAM-intensive kernels, we selected the top 4 kernels with the highest DRAM bandwidth utilization. For the kernels with low bandwidth utilization, they leverage a large amount of threads/warps to hide the memory latency. Note that even with a high number of warps, their overall bandwidth usage is low since either they are compute intensive or they are limited by other resources such as shared memory or branch divergence, which constrain their memory requests. However, when such kernels co-run with bandwidth-intensive kernels, the memory latency will be significantly increased such that higher levels of TLP are required to fully hide the latency. On the other hand, as the static resources are shared with all co-running kernels, each kernel may have fewer threads/warps to run concurrently on each SM to hide the memory latency. As a consequence, they become sensitive to increased latency and we refer to such kernels, i.e., those with low bandwidth utilization, as latency sensitive. To select the 5 kernels which are most sensitive to the increased memory latency, we doubled the access latency to L2 cache and DRAM and selected the ones with the most performance slowdowns.

Table 2 lists the details of the selected 13 benchmarks along with their resource utilization in standalone execution. The throughput (labeled 'Thro.' in Table 2) utilization is defined as the ratio of the achieved IPC (instruction per cycle) over the peak IPC. The NoC/DRAM bandwidth utilization is the ratio of the achieved bandwidth over the corresponding peak bandwidth. For the NoC-intensive kernels, we observe the NoC bandwidth utilization is saturated around 50% − 60% of the peak bandwidth. This is because the sustainable bandwidth of the crossbar NoC is 60% as pointed out in prior works [25, 35]. For the DRAM-intensive kernels, the DRAM sustainable bandwidth saturates around 60% − 70% of the peak bandwidth, which is consistent with the observation by X. Mei et al. [37]. When a kernel has bandwidth utilization close to the sustainable bandwidth, the memory system latency is significantly increased due to queuing delays. For example, the DRAM access latency in SAD becomes 6.8x higher than the idle latency, i.e., the access latency when the system is idle.

In the latency-sensitive category, BP and PF are compute intensive and they achieve high throughput. BFS and BT are thread-divergent kernels. The dynamic resources of HG are low because it is limited by the shared memory size. These kernels are latency-sensitive because they suffer from higher than 1.7x performance slowdown when the L2 and DRAM latency is doubled. Because the memory bandwidth is underutilized, the average L2/DRAM latency for these kernels is close to the idle latency.

In summary, we list the definitions of our kernel categorization as follows.

- Bandwidth-intensive (BI) kernels: kernels that are sensitive to the NoC or DRAM bandwidth. BI kernels include both NoC-intensive and DRAM-intensive kernels.
- NoC-intensive (NBI) kernels: kernels that are sensitive to the NoC bandwidth.
- DRAM-intensive (DBI) kernels: kernels that are sensitive to the DRAM bandwidth.
- Latency-sensitive (LS) kernels: kernels that are sensitive to the increased memory latency or the static resource capacity. LS kernels include both compute-intensive kernels and static-resource-limited kernels.
- Compute-intensive kernels: kernels that are limited by the ALU resources.
- Static-resource-limited kernels: kernels that are limited by the static resources, including register file, shared memory and the number of threads.

3.3 Evaluation Metrics

We evaluate the performance of CKE using two metrics: harmonic speedup (HSpeedup) [36] and weighted speedup (WSpeedup) [15]. The HSpeedup is a balanced metric for both system throughput

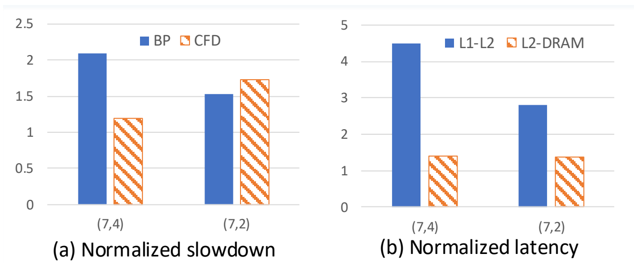


Fig. 2. Performance and latency impact for BP+CFD co-run using different CTA combinations.

and fairness [36] whereas WSpeedup is a metric for system throughput. The higher these metrics, the better performance is achieved. Because the goal of this paper is to improve both system throughput and fairness, HSpeedup is used as our primary evaluation metric. We also observe that these two metrics correlate well. The reason is that when the co-running kernels stress different resources, higher overall resource utilization means both higher and more balanced utilization, thereby higher throughput and better fairness.

$$HSpeedup = \frac{N}{\sum_k \frac{IPC_k^{alone}}{IPC_k^{share}}}$$

$$WSpeedup = \sum_k \frac{IPC_k^{share}}{IPC_k^{alone}}$$

To ensure our simulation sufficiently captures co-running kernels' dynamic behavior, we run 100 million cycles in each simulation experiment. If one kernel finishes before the 100 million cycles, it is re-executed. Because the IPCs of our benchmarks vary from 10s to 1000s. Therefore, the range of simulated instructions is 1 to 100 billions.

4 LIMITATIONS OF CTA MANAGEMENT

A common way to improve GPU CKE is to manage the CTA combination of co-running kernels [43, 52, 55]. Although appropriate CTA combinations throttle the TLP of bandwidth-intensive kernels and therefore their memory requests, such approaches are not sufficient to address the memory interference problem. The reasons include (1) the bursty nature of memory traffic, (2) the inability to prioritize memory requests from different kernels, and (3) coarse-grain TLP control at the CTA level. Next we dissect these effects.

4.1 Effects of Bursty Memory Requests

High memory access latency First, we use a case study of the BP+CFD co-run to illustrate the impact of bursty memory traffic on the memory access latency. BP is latency sensitive and CFD is NoC intensive. These two kernels are considered complementary as they stress different GPU resources. Through an exhaustive search of all possible CTA combinations, we find (7,4), meaning 7 CTAs from BP and 4 CTAs from CFD co-running on each SM, achieves the highest weighted speedup and harmonic speedup. Figure 2(a) shows the slowdown of BP and CFD in the co-run, normalized to their standalone execution using this CTA combination. Normalized slowdown is computed as IPC_{alone}/IPC_{shared}. Even though BP has higher CTA utilization than CFD, it suffers from higher performance slowdown when the CTA combination (7,4) is used. Figure 2(b) shows the normalized memory access latency, comparing to the BP standalone execution in this case. We can see that the L1-L2 latency is much higher due to the NoC congestion caused by CFD. Because neither

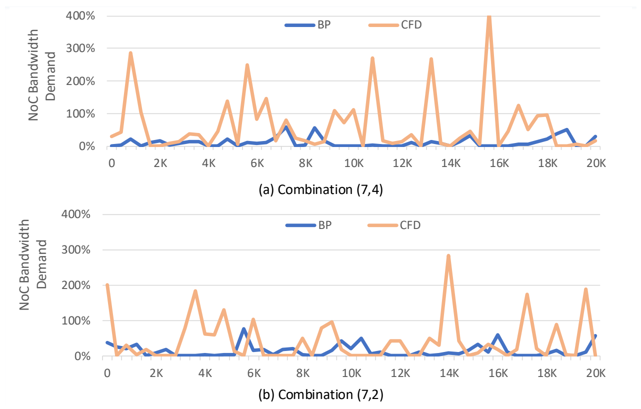


Fig. 3. The instantaneous NoC bandwidth demand of BP+CFD co-runs with different CTA combinations.

BP nor CFD is DRAM intensive, the L2-DRAM latency is not significantly increased. Regardless the high latency, we notice that 4 CTAs of CFD only consume 62% NoC bandwidth on average.

To pinpoint the reason for the increased L1-L2 latency, Figure 3 shows the instantaneous NoC bandwidth demand, which is normalized to the NoC sustainable bandwidth. To measure the bandwidth demand, we count the number of memory requests in the L1 D-cache miss queue to be sent to the NoC every 400 cycles. The request rate can be transformed to the bandwidth demands using the derivations to be discussed in Section 5.3. We call the bandwidth as 'over-subscribed' when the demand is higher than 100% of sustainable bandwidth. In such scenarios, the requests have to be lined up in the memory queues. In Figure 3(a), we can see the bursty memory traffic of CFD leads to frequent bandwidth over-subscription, resulting in very high queuing delays in the memory system. As a result, the memory requests from BP suffer from the long access latency.

Reducing the CTA number of bandwidth-intensive kernels can improve the performance of the co-running latency-sensitive kernels. For example, as shown in Figure 2(b), the performance of BP can be improved if we reduce the CTA number of CFD from 4 to 2. However, Figure 2(b) shows that the memory access latency is still much higher than it when BP runs alone. The average NoC bandwidth consumption is reduced to 44% with 2 CTAs from CFD. However, as seen in Figure 3(b), the bandwidth over-subscription problem remains due to the bursty behavior of the CTAs from CFD. On the other hand, reducing the CTA number of CFD results in the overall bandwidth being underutilized. Therefore, the combination of (7,2) shows worse weighted speedup than (7,4) (weighted speedup of 1.2 vs. 1.3).

Resource underutilization When an NoC-intensive kernel co-runs with a DRAM-intensive kernel, the GPU utilization is maximized when both NoC bandwidth and DRAM bandwidth are fully utilized. However, the bursty behavior may cause the memory system being dominated by either NoC or DRAM demands in a certain period, leading to underutilized bandwidth of the other. We illustrate such a phenomenon by co-running an NoC-intensive kernel, CFD, and a DRAM-intensive kernel, FTD. We find that the optimal CTA combination of CFD+FTD co-run is (4,7) through an exhaustive search. Figure 4 shows the instantaneous DRAM bandwidth utilization for 7 CTAs of FTD running alone and CFD+FTD co-run. The sampling period is 400 cycles. As we can see from the figure, the co-run shows lower but more bursty DRAM bandwidth consumption than FTD running alone. This is because CFD has a lot of L1 D-cache misses (which would hit in the L2 cache and therefore stress the L2-to-L1 NoC) in certain periods. Those requests dominate the memory queues between L1 and L2 caches so that FTD fails to issue requests. As a result, the DRAM bandwidth is wasted/underutilized in such periods even though FTD has DRAM bandwidth requests to be issued.

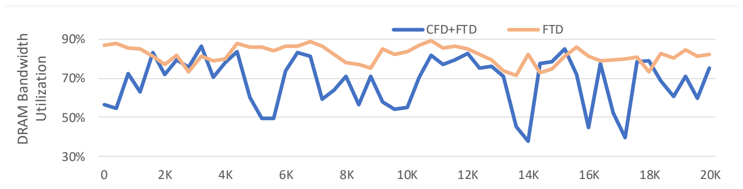


Fig. 4. Instantaneous DRAM bandwidth utilization for CFD+FTD co-run and FTD standalone execution.

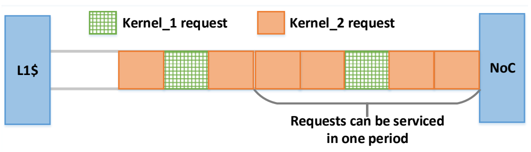


Fig. 5. A snapshot of the L1 D-cache miss queue when a latency-sensitive kernel (kernel_1) co-runs with a bandwidth-intensive one (kernel_2).

4.2 Prioritization of Memory Requests

We observe that the performance of CKE can be significantly improved if we can prioritize the memory requests from different kernels based on their different latency-hiding characteristics. For example, Figure 5 shows a snapshot of the L1 D-cache miss queue in a 2-kernel co-running case, where kernel_1 is latency sensitive and kernel_2 is bandwidth intensive. As kernel_2 typically generates much more requests than kernel_1, in this snapshot, kernel_1 has 2 requests while kernel_2 has 6 requests waiting in the miss queue. We assume that the critical bandwidth resource can only service 5 requests in a certain period. Given the FIFO policy of the miss queue, 1 request of kernel_1 and 4 requests of kernel_2 can be serviced in the first period. However, if we could change the policy to prioritize the latency-sensitive kernel, 2 requests of kernel_1 and 3 requests from kernel_2 would be serviced instead. The throughput improvement of kernel_1 could be as high as 100% while the throughput of kernel_2 is only reduced by 25%. Such memory request prioritization is hard to achieve with CTA-level TLP management.

4.3 Coarse Granularity of TLP Control

Another limitation of CTA combination is the coarse granularity of TLP management. For some kernels, even one CTA can cause high bandwidth utilization and long memory access latency. For example, a single CTA of the SAD kernel utilizes 89% DRAM bandwidth and causes 2.3x higher memory access latency than the idle latency. More often, adding one CTA of the bandwidth-intensive kernel may lead to congested memory systems while reducing one CTA may result in underutilized bandwidth. Therefore, the CTA-level management alone is likely to be too coarse-grained to achieve optimal system throughput or fairness.

4.4 Motivation of Our Approach

This section illustrates that, without considering the bandwidth impact of the co-running kernels, the CTA management alone is insufficient to achieve the optimal performance. In order to solve the bandwidth over-subscription problem due to the bursty memory traffic, we propose to strictly control the NoC and DRAM bandwidth consumption of each kernel. To determine the optimal CTA combination and bandwidth partition, our approach detects the characteristics of each kernel and allocates CTA and bandwidth resources accordingly.

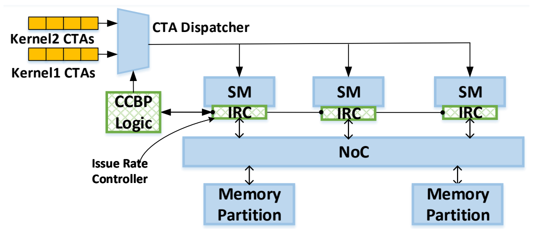


Fig. 6. An overview of the CCBP architecture.

5 COORDINATED CTA COMBINATION AND BANDWIDTH PARTITIONING

5.1 Overview

In this paper, we propose a coordinated CTA combination and bandwidth partitioning (CCBP) approach to improve the concurrent kernel execution on GPUs. Figure 6 presents the architecture of our CCBP approach. The CCBP logic executes the CCBP algorithm to determine the CTA combination and bandwidth partitioning for each kernel. It monitors the bandwidth consumption and achieves bandwidth partition through the issue rate controllers (IRCs). The CTA combination is controlled through the CTA dispatcher.

The issue rate controllers (IRCs) in Figure 6 determine the bandwidth consumption of both NoC and DRAM. We manage the bandwidth demands by controlling the memory request issue rate, i.e., how many memory requests to be issued per time unit (200 cycles in this paper), for each kernel. Depending on the memory request type (load or store) and whether it hits the L2 cache, a request may result in different consumption of NoC and DRAM bandwidth. As will be discussed in Section 5.3, our approach collects some factors of a kernel to derive the average NoC and DRAM consumption of a memory request and then achieve the overall NoC and DRAM bandwidth partitioning by managing the request issue rate.

There are three possible locations to place the IRC in an SM: the warp scheduler, the load-store unit, and the L1 D-cache miss queue. The warp scheduler cannot control the bandwidth consumption accurately because it is not aware of the memory coalescing effect. For the option of controlling the request rate at the load-store unit, the accesses, which turn out to be L1 D-cache hits, can be affected even they have no impact on the NoC (or DRAM) bandwidth consumption. Therefore, our choice is to control the issue rate of each L1D miss queue, i.e., how many L1D misses are sent to the L1-to-L2 NoC per time unit.

Based on the kernel type detection and the feedback from the IRCs, the CCBP algorithm determines the optimal CTA combination and bandwidth partitioning. Our key insight is that different types of kernels have different 'dominant' resources. The dominant resource of the latency-sensitive kernels is the CTA number because they require higher levels of TLP to hide the memory latency. For the NoC- or DRAM-intensive kernels, the dominant resource is the NoC or DRAM bandwidth, respectively.

5.2 CCBP Algorithm

As observed in Section 4, if we let the co-running kernels freely compete for the memory resources, the overall bandwidth demand may exceed the peak bandwidth. The resulting high queuing delays will significantly affect latency-sensitive kernels. However, if we evenly partition the bandwidth, some CTA and bandwidth resources may be wasted. So, there are three objectives of CCBP: (1) fully utilize both the CTA and bandwidth resources; (2) avoid bandwidth over-subscription by the bandwidth-intensive kernels; (3) fairly distribute the resources among the co-running kernels based on their dominant resources.

```
Data: $type_{1..K}$: kernel types;
      $nbw_{1..K}[1..N]$: NoC BW util with varied CTA numbers;
      $dbw_{1..K}[1..N]$: DRAM BW util with varied CTA numbers;
      $priorLSK$: priority for latency-sensitive kernels;
Result: $CTA_{1..K}$: number of CTAs for each kernel;
        $NBW_{1..K}$: NoC bandwidth quota;
        $DBW_{1..K}$: DRAM bandwidth quota;
$CTA_{1..k} \leftarrow \{0\}, NBW_{1..k} \leftarrow \{0\}, DBW_{1..k} \leftarrow \{0\}\ dom_{1..k} \leftarrow \{0\}$;
$unFinished \leftarrow \{1..K\}$;
while find k with lowest $dom_k$ in unFinished do
    ⊳ Allocate resources tentatively
    if $type_k = Latency\_Sensitive$ then
        $tmpCTA_k \leftarrow CTA_k + 1$;
        $tmpNBW_k \leftarrow nbw_k[tmpCTA_k]$;
        $tmpDBW_k \leftarrow dbw_k[tmpCTA_k]$;
    else if $type_k = NoC\_Intensive$ then
        $tmpNBW_k \leftarrow NBW_k + 1$;
        if $nbw_k[CTA_k] < tmpNBW_k$ then
            $tmpCTA_k \leftarrow CTA_k + 1$;
        else
            $tmpCTA_k \leftarrow CTA_k$;
        $tmpDBW_k \leftarrow dbw_k[tmpCTA_k] \times \frac{tmpNBW_k}{nbw_k[tmpCTA_k]}$;
    else
        $tmpDBW_k \leftarrow DBW_k + 1$;
        if $dbw_k[CTA_k] < tmpDBW_k$ then
            $tmpCTA_k \leftarrow CTA_k + 1$;
        else
            $tmpCTA_k \leftarrow CTA_k$;
        $tmpNBW_k \leftarrow nbw_k[tmpCTA_k] \times \frac{tmpDBW_k}{dbw_k[tmpCTA_k]}$;
    ⊳ Check if there are enough resources
    if CanAlloc(tmpCTA, tmpNBW, tmpDBW) then
        $CTA_k \leftarrow tmpCTA_k$;
        $NBW_k \leftarrow tmpNBW_k$;
        $DBW_k \leftarrow tmpDBW_k$;
        $dom_k \leftarrow DomShareCal(k, CTA_k, NBW_k, DBW_k, priorLSK)$;
    else
        $unFinished \leftarrow unFinished - k$;
end
```

Algorithm 1: CCBP algorithm based on DRF.

To achieve the objectives, our CCBP algorithm, as shown in Algorithm 1, leverages the 'Dominant Resource Fairness' (DRF) [16] approach to assign the CTA and bandwidth resources to different kernels. Compared to SMK [52], which determines the CTA combination using the DRF algorithm on static resources, our CCBP approach takes both static resources and dynamic bandwidth resources into account. In Algorithm 1, the *type* input contains the kernel type of each kernel. In Section 5.4, we will describe how the CCBP approach classifies the kernels into different types, i.e., latency sensitive, NoC intensive and DRAM intensive. The *nbw* and *dbw* inputs contain the NoC and DRAM bandwidth utilization and the computation of such utilization is described in Section 5.4. The *priorLSK* input is used to adjust the priority of the latency-sensitive kernels, which will be discussed in Section 5.5.

The dom_k value maintains the dominant resource utilization for kernel k. If the kernel is latency sensitive, the dominant resource are the static resources (register file, shared memory or thread number) that limit the CTA number. On the other hand, the dominant resource is the NoC or DRAM bandwidth for NoC- and DRAM-intensive kernels, respectively.

Similar to previous GPU intra-SM sharing approaches [11, 42, 52, 55], in CCBP, all SMs share the same CTA combination. And the bandwidth allocation for each kernel is evenly distributed to each

Table 3. An example of CCBP algorithm. Assume that, without bandwidth management, the CTA/NoC/DRAM consumption ratios for kernels K1, K2 and K3 are 2:1:1, 1:2:4 and 1:3:1.

	Iter 1			Iter 2			Iter 3			Iter 6			Iter 11			Iter 12			Iter 16		
	$\underline{\text{K1}}$	K2	K3	K1	$\underline{\text{K2}}$	K3	K1	K2	$\underline{\text{K3}}$	K1	K2	$\underline{\text{K3}}$	K1	$\underline{\text{K2}}$	K3	K1	K2	$\underline{\text{K3}}$	$\underline{\text{K1}}$	K2	K3
CTA Number	$\underline{1}$	0	0	1	1	0	1	1	1	2	1	1	4	1	1	4	1	2	$\underline{6}$	2	2
NoC Quota	0.5	0	0	0.5	0.5	0	0.5	0.5	$\underline{1}$	1	1	$\underline{2}$	2	2	3	2	2	$\underline{4}$	2.5	2.5	5
DRAM Quota	0.5	0	0	0.5	$\underline{1}$	0	0.5	1	0.33	1	2	0.66	2	$\underline{4}$	1	2	4	1.33	2.5	5	1.66

SM. The reason is that the CTAs in the same kernel typically exhibit very similar characteristics, comparing to the CTAs in different kernels.

The algorithm works as follows. Initially, resources assigned to all kernels are set to zeros. Each iteration in Algorithm 1 finds a kernel with the lowest dom_k value. Then the dominant resource allocated to that kernel is tentatively increased by 1 unit. For latency-sensitive kernels, as their dominant resource is the CTA number per SM, 1-unit resource means the static resources needed to accommodate one CTA on every SM. For NoC and DRAM bandwidth sensitive kernels, we evenly divide the dynamic bandwidth resource into M units. And 1-unit resource means 1 unit of dynamic bandwidth. The higher M, the finer granularity of bandwidth control. We find that when M is higher than 20, the algorithm output as well as the overall performance has very little difference. Therefore, we set M as 20 in this paper. For bandwidth-intensive kernels, their CTA numbers are increased only when the assigned bandwidth exceeds the overall bandwidth demand of the current CTA number. In each iteration of resource allocation, CCBP checks whether the total static resource and dynamic bandwidth available can accommodate such allocation. If so, the resources are allocated accordingly and the dom_k is updated using the function *DomShareCal*, which is described in Section 5.5. Otherwise, the resource allocation is finished.

Table 3 shows an example of the CCBP procedure on three kernels, K1, K2 and K3. Assume that K1 is latency sensitive and each CTA of K1 consumes 0.5 NoC bandwidth unit and 0.5 DRAM unit. K2 is DRAM intensive and we assume that each CTA of K2 consumes 2-unit NoC bandwidth and 4-unit DRAM bandwidth. K3 is NoC intensive and each CTA consumes 3-unit NoC bandwidth and 1-unit DRAM bandwidth. We assume each type of resource has a total of 10 units. In each iteration of CCBP, the algorithm allocates one unit of the dominant resource for the kernel with the minimum dominant share. Table 3 shows the status after a certain number of iterations. In each iteration, we highlight the kernel which is selected to increase its resources and its dominant resource. In the first iteration, when all kernels has zero dominant shares, CCBP selects the latency-sensitive kernel K1 and increases its CTA number by 1. And its NoC and DRAM quotas are increased proportionally to 0.5. In iteration 2, K2 is selected and its dominant resource, i.e., the DRAM bandwidth, is increased by 1 unit and the NoC bandwidth quota is increased proportionally by 0.5 unit. Similarly, the NoC quota of K3 is increased in iteration 3 to 1 unit and its DRAM quota is increased proportionally to 0.33 unit. At the end of iteration 11, the NoC-intensive kernel K3 has the minimum dominant share (3 units of NoC bandwidth) compared to the other two kernels. So, in iteration 12, its NoC quota is increased to 4. Because 1 CTA of K3 is not able to consume 4 units of NoC bandwidth, its CTA number is increased by 1. The CCBP algorithm terminates at iteration 16 when no more CTAs can be allocated. At this time, we can see that the dominant resources of all kernels have been fairly allocated.

After the resource allocation for all kernels is finished, CCBP returns the newly determined CTA combination and bandwidth quotas. Then, the static resources are re-partitioned to accommodate the new CTA combination. The assigned bandwidth quotas are converted to the L1 D-cache miss request issue rate quotas using the bandwidth management approach discussed in Section 5.3.

The CCBP algorithm can also be applied for kernels with more than one limiting resource. For example, a kernel can have similar NoC and DRAM resource utilization and both are limiting resources. In this scenario, the dominant resource can be detected as either limiting resource. With

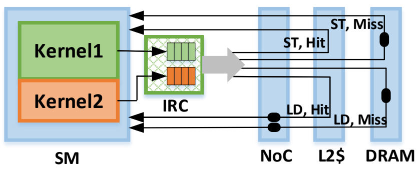


Fig. 7. 4 memory transaction types, black dots represent the bandwidth is consumed at a particular component.

the CCBP algorithm, whenever one share of the dominant resource is allocated, a proportional quota in the other resources, including the other limiting resources, are allocated accordingly. Therefore, the algorithm remains effective.

5.3 Bandwidth Management

We first illustrate some of the key concepts of our approach using Figure 7.

Figure 7 shows 4 types of memory transactions, load or store and hit or miss in the L2 cache. The black dots in NoC and DRAM denote that the bandwidth is consumed by such a transaction. For stores hit in the L2 cache, as we focus on the bandwidth consumption of NoC in the L2-to-SM direction and the size of store acknowledgements is small, we ignore the NoC bandwidth consumption by such store transactions. For a load transaction, the NoC bandwidth is consumed no matter it hits or misses the L2 cache. So, the NoC bandwidth consumption is proportional to the number of replied L1 cache lines per time unit, which equals to the number of load requests per time unit. The DRAM bandwidth is consumed only when a load/store transaction misses the L2 cache. Therefore, the DRAM bandwidth consumption is proportional to the product of the request issue rate and the average DRAM transactions per request. In our approach, we collect the statistics of the average DRAM transactions per memory transaction in the L2 cache. And such statistics is piggybacked in the replied packets. This architectural support will be discussed in Section 6.

Next, we derive the relationship between the L1 D-cache miss queue issue rate and the NoC/-DRAM bandwidth consumption.

For kernel k, given the L1 D-cache miss request issue rate $IssueRate_k$ (issued requests per time unit), the L2-to-SM NoC bandwidth consumption $BWReq_n^k$ can be derived as:

$$\begin{aligned} BWReq_n^k &= L2L1ReplyRate \times \#L1Caches \times ReplySize \\ &= L1L2ReadRequestRate \times \#SMs \times ReplySize \\ &= IssuedReadReqPerTimeUnit \times \#SMs \times ReplySize \\ &= rf_k \times IssueRate_k \times \#SMs \times (L1BlkSize + FlitSize) \end{aligned} \tag{1}$$

In which, rf_k is defined as the ratio of the read requests over the total memory requests:

$$rf_k = \frac{\#ReadRequests_k}{\#Requests_k} \tag{2}$$

Because we only consider the NoC bandwidth consumption by load transactions, the number of transactions sent to each SM is $rf_k \times IssueRate_k$ per time unit. Each transaction or packet through the NoC includes the payload and a header. The payload is the L1 D-cache block replied from the L2 cache. The header, which occupies a flit, contains the request meta data such as the destination SM id and warp id.

Likewise, the DRAM bandwidth consumption for kernel k can be derived as a function of the L1 D-cache miss queue issue rate $IssueRate_k$:

$$BWReq_d^k = df_k \times IssueRate_k \times L2BlkSize \times \#SMs \tag{3}$$

In which, df_k denotes the number of DRAM accesses generated per L2 cache request:

$$df_k = \frac{\#DRAMAccesses_k}{\#Requests_k} \tag{4}$$

Therefore, $df_k \times IssueRate_k$ is the number of DRAM accesses per time unit. Note that df_k has a subtle difference from the L2 cache miss rate. The reason is that when an L2 cache request misses the L2 cache and the victim cache block is dirty, the number of DRAM accesses is 2. The DRAM access granularity is *L2BlkSize* per request in our model.

From Equations 1 and 3, we can see that the factors, rf and df,of each kernel are the keys to control the NoC and DRAM bandwidth consumption using the request issue rate. Therefore, before partitioning the NoC/DRAM bandwidth, rf and df need to be collected using Equations 2 and 4, which can be done during the kernel-type detection step.

5.4 CCBP-Parameter Detection

In this section, we discuss how the input parameters of CCBP algorithm are detected. These input parameters include the kernel types and the relationship between the CTA number and the corresponding NoC/DRAM bandwidth utilization.

Even Partitioning

The input parameters are detected using even CTA combination and bandwidth partitioning. Under even CTA combination, the CTA number of each kernel is set to $\#AloneRunCTA_k/\#Kernels$. Then, the NoC and DRAM bandwidth are also partitioned evenly, which means to assign each kernel an even share of the NoC and DRAM bandwidth. To achieve even partitioning, the $IssueRate_k$ of each kernel is set to the maximum value that satisfies the even-share conditions, which can be represented as:

$$\begin{aligned} IREven_k = Max(IssueRate_k) : BWReq_n^k &\leqslant \frac{SustainBW_n}{\#Kernels}, \\ BWReq_d^k &\leqslant \frac{SustainBW_d}{\#Kernels} \end{aligned} \tag{5}$$

The issue rate under such conditions can be solved using Equations 1 and 3 and is named $IREven_k$ for kernel k. For example, between 2 co-running kernels, either is assigned with 1/2 sustainable NoC bandwidth and DRAM bandwidth. Such an allocation is then converted to two issue rates using Equations 1 and 3 respectively. The $SustainBW_n$ and $SustainBW_d$ denote the NoC and DRAM sustainable bandwidth. As discussed in Section 3, in our model, the NoC and DRAM sustainable bandwidth are 60% and 70% to their theoretical peak bandwidth. When other types of NoC or memory are used for GPU, these two parameters need a one-time update to reflect the sustainable bandwidth on the corresponding technologies.

Kernel-Type Detection

With the issue rates of all kernels being set based on Equation 5, we decide a kernel as bandwidth intensive if its average memory request demand has an equal or higher rate than the even-partitioned issue rate, i.e., $IRDemand_k >= IREven_k$. Otherwise, the kernel is considered latency sensitive.

A bandwidth-intensive kernel can be further classified as NoC intensive or DRAM intensive based on its relative NoC and DRAM bandwidth utilization. Specifically, a kernel is NoC intensive if:

$$\frac{BWReq_n^k}{SustainBW_n} > \frac{BWReq_d^k}{SustainBW_d} \tag{6}$$

Take Equations 2-3 into Inequation 6, we get:

$$\frac{df_k}{rf_k} < \frac{SustainBW_d}{SustainBW_n} \times \frac{L1BlkSize + FlitSize}{L2BlkSize} \tag{7}$$

If this inequality holds, the kernel is NoC intensive. Otherwise, it is DRAM intensive. Intuitively, df_k in Inequation 7 is positively correlated to the L2 cache miss rate. **If the L2 cache miss rate is low meaning that a substantial amount of requests are filtered, the NoC becomes the bottleneck in the memory system. Otherwise, the DRAM is the main resource to reply the requests and the DRAM bandwidth is the bottleneck.** This conclusion can be confirmed from Table 2, which shows that NoC-intensive kernels have much lower L2 miss rates than DRAM-intensive kernels.

Bandwidth-Utilization Detection

Once the kernel types are detected, our CCBP approach collects the bandwidth resource demand/utilization with different CTA numbers, i.e., the *nbw* and *dbw* inputs in Algorithm 1, for each kernel.

For latency-sensitive kernels, CCBP assumes the bandwidth utilization scales linearly with the CTA number. Therefore, it leverages the average bandwidth utilization per CTA to calculate the bandwidth utilization for different CTA numbers. For a bandwidth-intensive kernel, CCBP learns its bandwidth utilization with different CTA numbers using a throttling approach. First, it releases the bandwidth limitation for this kernel and throttles a certain number of CTAs in this kernel. When a CTA is throttled, the warps in the CTA cannot be selected by the warp scheduler to issue any instructions. Under such circumstances, the bandwidth utilization is collected for this particular number of active CTAs.

Static or Dynamic Detection

In our approach, the CCBP parameters can be either detected online or offline. In order to capture the interference among the co-running kernels, either online or offline detection is performed under co-running environment with even CTA combination.

With online detection, our approach skips the first 50K cycles for warming up. The next 20K cycles are used to collect the rf and df factors as defined in Section 5.3. Then, the NoC and DRAM bandwidth can be partitioned evenly using Equation 5. Under even CTA combination and bandwidth partitioning, we determine the kernel types as described above. To detect the bandwidth utilization with varied CTA numbers of a bandwidth-intensive kernel, we release the bandwidth limitation of such a kernel and throttle its active CTA number. For each CTA number of interest, we skip the 10K cycles and use the next 10K cycles to collect the bandwidth utilization.

In offline detection, the procedure is the essentially same as online detection except that we use 10x longer detection time in each step to improve the accuracy.

5.5 CCBP with Adjustable Priorities

Although the CCBP algorithm as described in Section 5.2 fairly allocates the resources, we observe that the latency-sensitive kernels can still suffer from higher performance slowdown when co-running with bandwidth-intensive kernels. The reason is that although our approach can significantly reduce the memory latency in the CKE environment, it is still higher than the latency when a latency-sensitive kernel runs alone. In order to achieve the fairness as well as high throughput, the CCBP approach offers higher priority to latency-sensitive kernels.

As shown in Algorithm 1, the *DomShareCal* function calculates the dominant resource utilization of a kernel. For a NoC-/DRAM-intensive kernel, the result is the ratio of assigned NoC/DRAM bandwidth and the total sustainable NoC/DRAM bandwidth, i.e.,

$$dom_{bwk} = \frac{AssignedDomBW_{bwk}}{SustainDomBW_{bwk}}$$

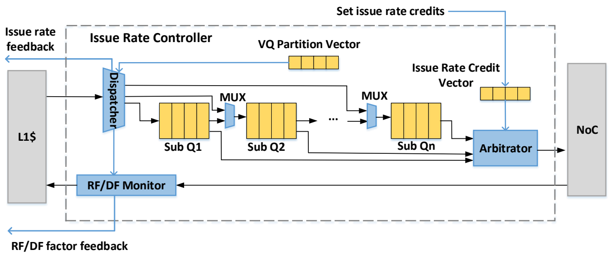


Fig. 8. Architecture of an issue rate controller (IRC).

For a latency-sensitive kernel, the result is scaled with the priority:

$$dom_{lsk} = \frac{AssignedDomStaticRes_{lsk}}{TotalDomStaticRes_{lsk}} \times priorLSK$$

The static resources of a kernel are the registers, shared memory and thread number and the dominant static resource is the resource with the highest utilization. The $priorLSK$ is the priority for the latency-sensitive kernels. Lower $priorLSK$ means higher priorities. When $priorLSK$ is 1.0, latency-sensitive kernels have equal priority as bandwidth-intensive kernels .

In our approach, the $priorLSK$ is set to 1.0 in the beginning. Then it is adjusted using a hill climbing algorithm and $priorLSK$ is decreased by 0.1 in each iteration. For each priority, the CCBP algorithm determines a certain resource assignment and monitors the performance while running for 100K cycles using such an assignment. The performance goal can either be Wspeedup or Hspeedup as shown in Section 3. The adjustment of $priorLSK$ stops when the performance reaches the highest value.

6 ARCHITECTURAL SUPPORT

In this section, we discuss how the issue rate controller (IRC) in Figure 6 is implemented. Figure 8 shows the architecture of an IRC. To monitor the bandwidth consumption, IRC collects memory request issue rate of each kernel. In order to transform the issue rate to the bandwidth consumption, the rf and df factors are monitored according to the Equations 2 and 4. After the CCBP algorithm determines the bandwidth partitioning, the corresponding issue rate quotas are sent to the IRC.

6.1 Virtual Queues

As shown in Figure 8, we implement the miss queue as a set of sub-queues between the L1 D-cache and the NoC such that each co-running kernel has its private queue, which is referred to as a virtual queue. The total number of the sub-queues equals the maximum number of kernels that an SM can accommodate. When the kernel number is less than the number of sub-queues, the sub-queues are evenly assigned to co-running kernels. When a memory request is generated (i.e., an L1 D-cache miss), it is sent to a certain virtual queue based on the kernel id of the request. A virtual queue partitioning vector is maintained to map the kernel ids to the tails of the virtual queues.

6.2 Request Arbitrator

The Issue Rate Credit Vector (IRCV) contains the number of credits assigned to each kernel. At each time interval (200 cycles in our model), the credits are reset to the issue rate quotas. In each cycle, the arbitrator only selects requests from kernels with non-zero credits. The credit of one kernel decreases by one when the arbitrator issues one request from this kernel. Among the kernels with non-zero credits, the arbitrator prioritizes the requests of latency-sensitive kernels. This way, memory requests from latency-sensitive kernels will not be delayed by those from bandwidth-intensive co-running kernels.

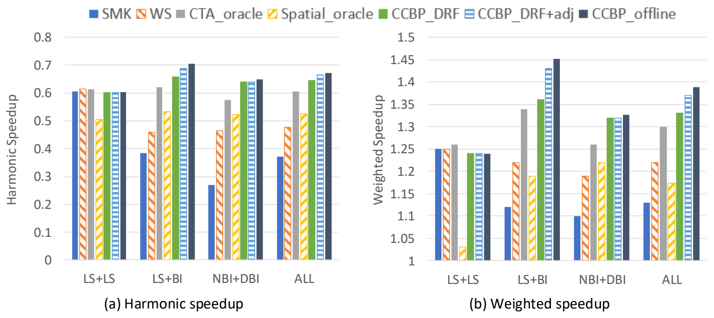


Fig. 9. Overall performance of 2-kernel co-runs.

In our evaluation, a 1-cycle latency is simulated for the arbitrator. The performance impact of the memory request arbitrator is negligible because it is not on the critical path. The NoC in our model can only transmit one 32-byte packet per cycle. However, one memory transaction contains a 128-byte cache block and a 32-byte header. So, the maximum number of requests can be issued in a 200-cycle time unit is 40. In our experiments, we vary the arbitrator latency from 1 to 4 cycles. The results show that the overall performance slowdown is only $0.11\% - 0.72\%$. The reason is that the added latency is only for memory accesses that miss the L1 D-cache. In addition, the L2 cache access latency is already high on GPUs even without queuing delays. And it becomes much higher when a bandwidth-intensive kernel is running, as shown in Figure 2. Therefore, the few-cycle arbitrator latency does not affect performance much.

6.3 Tracking and Monitoring DRAM Accesses for df factors

To retrieve the number of DRAM accesses for each request, we add a 2-bit field in the memory transaction header. As the NoC transmits the transactions in the unit of a flit, the header size is one flit (32B). So this extra 2-bit field does not increase the NoC bandwidth consumption.

We also add the following relatively simple logic in the L2 cache to track the number of DRAM accesses resulting from a request. For an L2 hit, the number is filled with 0. For an L2 miss which leads to a non-dirty block eviction, the number is 1. If the victim block is dirty, the number is 2. The DF monitor collects the DRAM access numbers in the replied transaction header to calculate the df factor defined in Equation 4.

6.4 Overall Hardware Cost

Sub-Queues We reuse the existing L1 D-cache miss queue to implement the sub-queues as shown in Figure 8.

Given the number of sub-queues N, which is also the number of maximum kernels in each SM, $3N - 1$ sets of wires and $N - 1$ multiplexers are added to implement the sub-queues.

Per Kernel Counters/Registers The Virtual Queue Partitioning Vector has N entries, each of which has $\log_2 N$ bits to point to the tail sub-queue of a kernel.

The Issue Rate Credit Vector has N entries and each entry has 8 bits as the maximum issue rate is 200 per interval.

In the RF monitor, two 32-bit counters per kernel are maintained to monitor the rf factor as defined in Equation 2. Similarly, in the DF monitor, two 32-bit counters are used to monitor the df factor as defined in Equation 4.

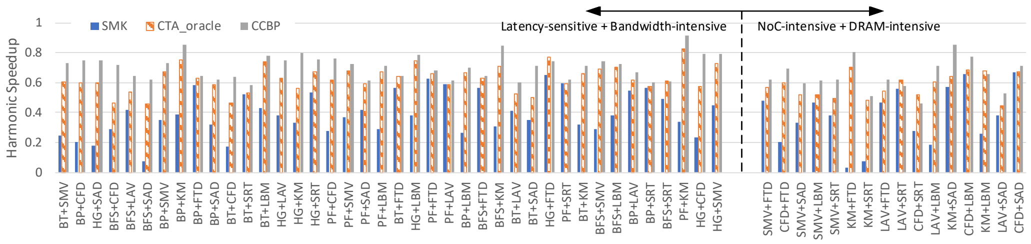


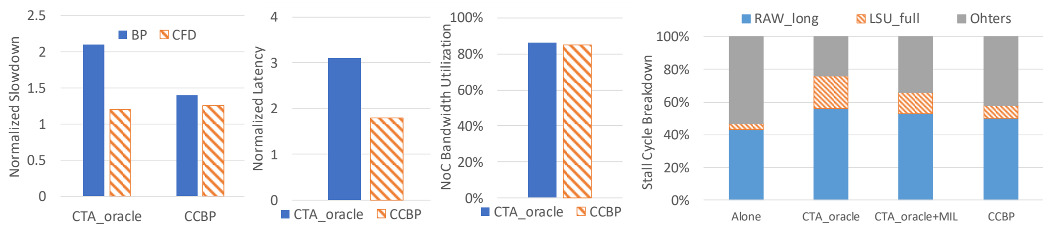
Fig. 10. Harmonic speedup of 2-kernel co-runs.

7 EVALUATION

7.1 2-Kernel Co-Runs

In this section, we evaluate all 2-kernel combinations under the categories of 2 latency-sensitive kernels (labeled as 'LS+LS'), 1 latency-sensitive and 1 bandwidth-intensive kernel (labeled as 'LS+BI'), 1 NoC-intensive and 1 DRAM-intensive kernel (labeled as 'NBI+DBI'). We do not combine two kernels that stress the same bandwidth resource because the overall throughput cannot be improved by such co-runs. Figure 9 shows the average results for each category using the geometric mean and the overall results are labeled as 'ALL'. In Figure 10, we report the harmonic speedup of each individual combination from 'LS+BI' and 'NBI+DBI' categories. SMK [52] is a prior work which leverages both CTA-level and warp-scheduler-level partitioning to improve CKE. WS [55] is another prior work which uses the scalability curves to determine the CTA combinations. The CTA_oracle is the best result through exhaustively searching all possible CTA combinations using intra-SM sharing. The spatial_oracle shows the best results using Spatial Multitasking [1, 51], in which SMs are evenly partitioned among co-running kernels. As pointed out by Wang et. al [51], the CTA combinations also have significant performance impact on Spatial Multitasking. Therefore, in Spatial_oracle, we also search all the possible CTA combinations to find the oracle one. Note that, for either CTA_oracle or Spatial_oracle, the optimal CTA combinations may be different for the highest harmonic speedup or the highest weighted speedup. CCBP_DRF is the CCBP approach without adjustable priorities for latency-sensitive kernels. CCBP_DRF+adj is the CCBP approach with adjustable priorities. For these two approaches, the inputs of CCBP algorithm are detected online using the method described in Section 5.4. CCBP_offline is the CCBP approach with the parameters and priorities of latency-sensitive kernels determined offline. In the subsequent sections, the CCBP_DRF+adj approach is used as default if not specified.

For the 'LS+LS' category, bandwidth partitioning has limited impact. So, our CCBP approach degrades to choosing the CTA combination by fairly partitioning the static resources, which is the same as the DRF approach used by SMK. For this category, we find that the performance of DRF static resource partitioning is very close to CTA_oracle. For Spatial_oracle, however, the performance is worse than intra-SM sharing approaches because it fails to improve the utilization of on-chip resources, such as register file and shared memory. For the 'LS+BI' category, our CCBP approach can significantly reduce the memory latency, which leads to the performance improvement of the latency-sensitive kernels. Compared to CTA_oracle, CCBP improves the harmonic speedup by 11% and improves the weighted speedup by 7% on average. For the 'NBI+DBI' category, the harmonic speedup and weighted speedup can be improved by 12% and 5% on average, respectively, over CTA_oracle. Compared to SMK and WS, CCBP improves the average harmonic speedup by 78% and 39% respectively. For the weighted speedup, the average improvements are 21% and 12% respectively. Compared to Spatial_oracle, the CCBP approach improves the harmonic speedup and weighted speedup by 28% and 18%, respectively.



(a) Normalized slowdown (b) Normalized latency (c) Noc bandwidth utilization (d) Stall cycle breakdown of warp scheduler for BP

Fig. 11. Case studies of BP+CFD.

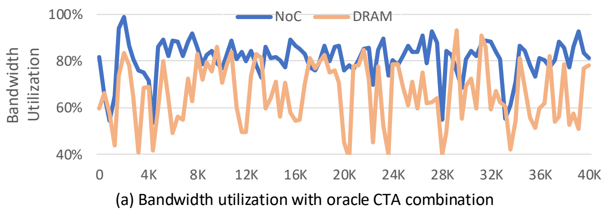
7.2 Impact on LS+BI Co-Runs

We use a case study, BP+CFD co-run, to illustrate the impact of CCBP when a latency-sensitive kernel (BP) co-runs with a bandwidth-intensive kernel (CFD). Figure 11(a) shows the performance slowdown of BP and CFD. With the oracle CTA combination, the latency-sensitive kernel BP suffers from much higher performance slowdown than the bandwidth-intensive kernel CFD. The reason is that the average memory access latency is significantly increased by the bandwidth-intensive kernel. Figure 11(b) shows the average memory access latency for BP, normalized to the latency of its standalone execution. With the CCBP approach, the memory access latency can be effectively reduced, leading to the performance improvement of the latency-sensitive kernel. On the other hand, the critical bandwidth utilization, as shown in Figure 11(c), is maintained at high levels. So the performance of the bandwidth-intensive kernel is not impaired significantly. The harmonic speedup for BP+CFD is improved by 25% with CCBP over CTA_oracle as shown in Figure 10.

For kernel BP, we illustrate the stall cycle breakdown of the warp scheduler in Figure 11(d). When a warp instruction is stalled, we accumulate the total stall cycles and attribute each stall into one of three reasons. If a dependent register is waiting for a memory fetch that missed the L1 D-cache, the stall is categorized as 'RAW_long' in the figure. If the instruction cannot be issued because the LSU is full, the stall cycle is categorized as 'LSU_full'. We categorize the stalls due to any other reasons as 'Others' in the figure. The figure shows the percentage of each category to the total stall cycles. When BP runs alone, 'RAW_long' accounts for 43% of the stall cycles and 'LSU_full' accounts for a very small portion. When BP co-runs with CFD, the percentage of 'RAW_long' is increased to 56% of the total stalls because the memory latency is increased. In addition, the percentage of 'LSU_full' is increased to 20% because the memory pipeline is dominated by CFD. With the CCBP approach, the 'RAW_long' is reduced to 49% because the memory latency is reduced. Besides, the 'LSU_full' percentage is reduced to 8% because the sub-queue structure can effectively prevent the memory pipeline being dominated by CFD. In this figure, we compare against the CTA_oracle with the MIL scheduling policy [11]. MIL throttles the memory instruction issuing when the L1 D-cache reservation failure rate is high and releases the limit when the rate is low. As shown in the figure, although MIL can also reduce the congestion of the memory pipeline, it is less effective than CCBP for the same reasons that are discussed in Section 7.5.

7.3 Impact on NBI+DBI Co-Runs

Figure 12 shows the instantaneous NoC and DRAM bandwidth utilization of CFD+FTD. CFD is NoC intensive while FTD is DRAM intensive. As shown in Figure 12(a), with the oracle CTA combination, both NoC and DRAM exhibit very bursty behavior. As pointed out in Section 4, over-subscribed NoC bandwidth in a period can cause underutilized DRAM bandwidth and vice versa. Because CCBP accurately controls both bandwidths, as shown in Figure 12(b), the burstiness of NoC and DRAM bandwidth utilization is significantly reduced. Compared to CTA_oracle, CCBP improves



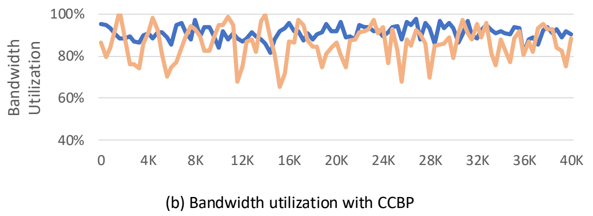


Fig. 12. Bandwidth utilization of CFD+FTD.

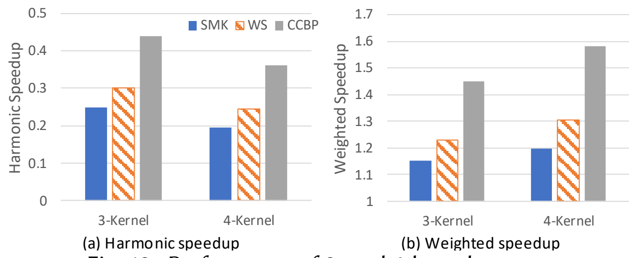


Fig. 13. Performance of 3- and 4-kernel co-runs.

the average NoC and DRAM bandwidth by 11% and 30%, respectively, leading to a 16% improvement in harmonic speedup, as shown in Figure 10.

7.4 3- and 4-Kernel Co-Runs

In Figure 13, we evaluate CCBP with 3 and 4 co-running kernels, each set has 40 randomly selected kernel combinations. Because the searching space for the oracle CTA combinations becomes too large, we choose to compare CCBP with prior works SMK [52] and WS [55]. For 3-kernel co-runs, CCBP improves the harmonic speedup by 46% and weighted speedup by 18% over WS. For 4-kernel co-runs, CCBP improves the harmonic speedup by 48% and weighted speedup by 21% over WS.

When there is more than one bandwidth-intensive kernel stressing the same critical bandwidth, the memory access latency is much more aggregated without bandwidth-aware management. Take HG+LBM+SAD as an example, because both LBM and SAD are DRAM intensive, the latency-sensitive kernel HG suffers from extremely high memory access latency. Compared to its standalone execution, the memory access latency is increased to 4.8x and 4.5x with SMK and WS, respectively. In contrast, with CCBP, the average memory access latency is only 40% higher than it in its standalone execution.

7.5 Comparison with Hybrid Approaches

We compare CCBP with four hybrid approach that integrates WS [55] CTA combination and different cache partitioning and memory scheduling approaches, including TCM [28], BMI [11], MIL [11], and UCP [44]. Figure 14 shows the harmonic speedup of these approaches normalized to the WS approach alone for 2-kernel co-runs.

TCM is an application-aware memory scheduling policy which prioritizes latency-sensitive applications in the DRAM memory controller. This approach is not effective for ‘LS+NBI’ and

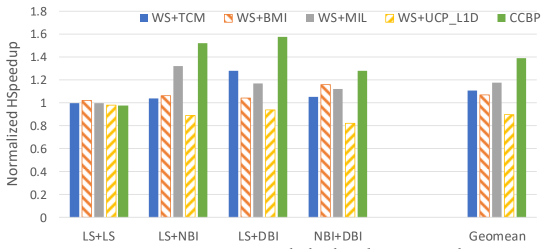


Fig. 14. Comparison with hybrid approaches.

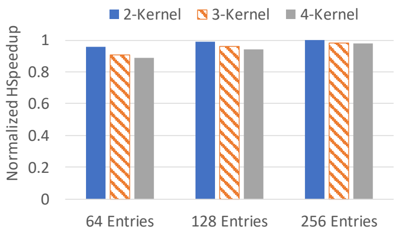


Fig. 15. Normalized harmonic speedups for varied queue sizes.

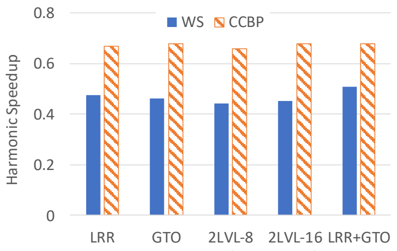


Fig. 16. Harmonic speedups with various warp scheduling policies.

'NBI+DBI' co-running kernels because it does not consider the NoC bandwidth utilization. For the 'LS+DBI' category, the memory request prioritization can improve the performance of latency-sensitive kernels through reducing the memory latency.

BMI and MIL are memory instruction scheduling policies for GPU CKE. Through controlling the memory instruction priority, BMI issues memory requests from co-running kernel in a balanced manner. BMI does not improve the performance for 'LS+NBI' and 'LS+DBI' kernels because it cannot effectively reduce the memory latency. MIL throttles the memory instruction issuing when the L1 D-cache reservation failure rate is high. Reservation failure happens when the L1 D-cache doesn't have enough resource to handle cache misses. In our experiments, we observe that NoC-intensive kernels have high L1 D-cache reservation failure rate. However, for DRAM-intensive kernels, the contention mainly occurs at L2-to-DRAM instead of L1 D-cache. Therefore, MIL is less effective for 'LS+DBI' than 'LS+NBI'. In contrast, our proposed CCBP is effective for both cases.

UCP is a cache partitioning method based on the cache way utility of each co-running application. It assigns more cache resources to an application if it is likely to reduce more cache misses. In our experiment, UCP is implemented on the L1 D-cache. As shown in Figure 14, in general, UCP has a negative performance impact on GPU CKE. The reason is that the L1 D-cache is entirely dominated by the bandwidth-intensive kernels without cache partitioning. With cache partitioning, the actual cache size for the bandwidth-intensive kernel is reduced, which further increases the bandwidth contention. This phenomenon is also observed by H. Dai et al. [11].

A common limitation of the hybrid approaches is that they are not able to control the bandwidth coordinately with the CTA combination. When the bandwidth is saturated, some CTA resources for the bandwidth-intensive kernels are wasted. In our CCBP approach, these CTA resources can be re-allocated to the latency-sensitive kernels to further improve their performance.

7.6 Sensitivity Studies

In this section, we analyze the sensitivity of CCBP to the sub-queue sizes and warp scheduling policies. In Figure 15, we vary the sub-queue size from 8 to 32 entries while fixing the number of sub-queues to 8. So the total queue size varies from 64 to 256. The figure shows the normalized harmonic speedup compared to the unlimited queue size. With 64-entry queues, the 2-, 3- and

4-kernel co-runs achieves 96%, 91% and 89% of the harmonic speedup with unlimited queue entries. With 128-entry queues, the achieved harmonic speedup is 99%, 96% and 94% respectively. And the harmonic speedup achieved with 256-entry queues is 100%, 98% and 98%. In general, more kernels concurrently running on each SM, more queue entries are needed to maintain high performance. For 2-kernel co-runs, the 64-entry queues are sufficient to achieve high harmonic speedups.

Figure 16 shows the harmonic speedups of the CCBP approach for 2-kernel co-runs with different warp scheduling policies. The LRR policy is used as the default in our experiments. The greedy-then-oldest (GTO) policy runs a single warp until it stalls then picks the oldest ready warp. In the two-level (2LVL) policy [39], the warps are divided into groups and it executes from only one group until all warps in that group are stalled. The round-robin policy is used for both intra- and inter- group scheduling. In Figure 16, '2LVL-x' denotes the two-level policy with group size x. Maestro [43] proposed the LRR+GTO scheduling policy for CKE. In this policy, the GTO warp scheduling policy is used within each kernel to increase the single kernel performance while the LRR policy is used to determine which kernel to schedule. With the WS CTA partitioning algorithm, the LRR+GTO scheduling policy achieves the highest harmonic speedup, which is 15% higher than the 2LVL-8 policy. On the other hand, we observe that with our proposed CCBP, different scheduling policies achieve similar performance. The reason is that the memory requests are fairly managed with CCBP regardless the warp scheduling policy.

8 RELATED WORK

GPU Concurrent Kernel Execution To support CKE on GPUs, Spatial Multitasking [1] partitions the GPU at the SM granularity, i.e., different kernels are assigned to different SMs. In order to improve the intra-SM resource utilization, SMK [52], WS [55] and Maestro [43] propose to dispatch the CTAs from different kernels to the same SMs and different approaches are proposed to determine the CTA combination. Based on intra-SM sharing, H. Dai et al. [11] propose memory instruction scheduling policies to reduce the memory pipeline stalls. The differences from these works have been discussed in Sections 4.

A recent work [51] proposes to control the TLP of each kernel through monitoring the effective bandwidth. Similar to Spatial Multitasking, their work evenly distributes the co-running kernels to different SMs, therefore the resources within an SM, including ALU, static resources and NoC ports, cannot be fully utilized. On the contrary, our work can achieve better resource utilization because we leverage intra-SM sharing and the resources are assigned according to the kernels' characteristics.

Jog et al. [22, 23] propose memory scheduling policies on the GPU memory controller. Similar to TCM [28], they do not address the congestion in the NoC or the queues between L1 and L2 caches. Li et al. [33, 34] propose a framework for CTA partitioning and cache bypassing based on off-line profiling. In comparison, our work dynamically detects the kernel type and partitions the critical resources accordingly. Targeting the single kernel running on GPUs, some memory scheduling approaches have been proposed to improve the GPU memory efficiency. Yuan et al. [56] propose a memory scheduling policy at the NoC to improve the DRAM row buffer locality. A DRAM scheduling has been proposed by Lakshminarayana et al. [29] to select between Shortest-Job-First (SJF) and FR-FCFS policies. Based on warp divergence characterization, a set of cache management and memory scheduling policies have been proposed by Ausavarungnirun et al. [5] to reduce the negative effects of memory divergence. Jog et al. [24] observe the latency tolerance of GPU cores are different for some applications and they propose a memory scheduling policy to prioritize the critical memory requests.

Besides the architectural support, some software approaches have also been proposed. Elastic kernel [42] and SM-Centric [54] are two compiler approaches to enable scheduling CTAs from different kernels on the same GPUs. Jiao et al. [21] propose a power-performance model to select concurrent kernels which can mostly improve the energy efficiency. Kernelet [58] divides a GPU kernel into multiple slices, each of which has variable occupancy to allow co-scheduling with other slices. Aguilera et al. [2] improve the fairness of Spatial Multitasking by balancing the individual performance and the overall performance. Ukidave et al. [49] extend the OpenCL runtime environment to explore several dynamic spatial multiprogramming approaches. Compared to the architectural approaches, the software approaches require source code modification.

Baymax [10] and Prophet [9] focus on system-level Quality of Service (QoS) support for GPUs. They leverage the PCIe data transfer model and kernel execution model to meet the QoS target. Wang et al. [53] assumes the QoS target of a kernel can be transformed to an IPC goal. In order to achieve the IPC goal, they propose to manage the warp scheduler quota and CTA combination. Our proposed accurate bandwidth allocation can be leveraged to provide more accurate QoS control.

Memory Scheduling on CMP or Heterogeneous Systems Numerous works have been proposed to improve memory scheduling on multi-core or chip-multiprocessor systems. Fair Queue [40] provides a fair allocation of the memory system to each thread regardless its memory demand. N. Rafique et al. [45] propose a bandwidth management scheme that periodically tunes the bandwidth to achieve desirable memory latency. PAR-BS [46] processes memory requests in batch to avoid unfairness and it exploits bank-level parallelism to achieve high throughput. ATLAS [27] prioritizes the threads that have attained least from the memory scheduler. TCM [28] groups the threads into latency-sensitive and bandwidth-intensive clusters and prioritizes latency-sensitive clusters. STFM [38], FST [14], MISE [48] and ASM [47] propose dynamic slowdown models to estimate the slowdown of each individual thread. Then, they prioritize the application that suffers from higher slowdown or throttle the one with lower slowdown. Their models cannot be directly used on GPUs because they don't consider the performance impact of the number of threads/CTAs. In addition, with their mechanisms, the thread/CTA number, i.e., TLP, is not managed to achieve the optimal performance for GPUs. MITT [59] provides a fine-grain memory traffic shaping mechanism for cloud systems.

On heterogeneous architectures, where the CPU and GPU share the last level cache (LLC) and off-chip memory, some works [4, 26, 30, 50, 57] observe that the heavy memory traffic from the GPU can be detrimental to the CPU. They propose to prioritize the CPU memory requests or throttle the GPU requests to improve the overall performance. In contrast, we focus on managing the memory traffic among the SMs within a GPU.

NoC Optimization on CMP or GPU systems On CMP architectures, the NoC has been widely studied [7, 12, 13, 18, 19, 31, 32]. GSF [31] proposes a QoS support for NoC by providing a minimum bandwidth and maximum latency guarantee. R. Das et al. [12, 13] propose application-aware prioritization mechanisms on NoC routers to improve system-level fairness and throughput. Some previous works [6, 20, 60] propose NoC optimizations for GPU systems. However, none of these prior works considers concurrent kernel execution, in which the bandwidth traffic is mixed from different kernels and the latency impact is also different for different kernels.

9 CONCLUSIONS

In this paper, we focus on improving GPU concurrent kernel execution. We observe that bandwidth over-subscription from bandwidth-intensive kernels results in high queuing delays in the GPU memory system, which leads to severe performance degradation for co-running latency-sensitive kernels. In addition, among bandwidth-intensive kernels, the memory bandwidth can also be unfairly consumed due to their different memory request intensities. To address these problems, we first make a case that the CTA-level management alone is insufficient and reveal the fundamental reasons. Based on these observations, we propose a coordinated approach to control CTA combination and bandwidth partitioning.

Our approach allocates the CTA and bandwidth resources based on kernel-type detection and resource requirement of each kernel. It assigns more CTA resources for the latency-sensitive kernels and assigns more bandwidth resources for the bandwidth-intensive kernels. Besides effective bandwidth utilization, the memory latency is highly reduced due to bandwidth control and burstiness reduction. The experimental results show significant improvements in both throughput and fairness, comparing to the oracle CTA combination and the state-of-the-art GPU concurrent kernel execution methods.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments. This work is supported by NSF grants, CCF-1618509, CNS-1717550, and an AMD gift fund.

REFERENCES

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. 2012. The case for GPGPU spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12.

[2] P. Aguilera, K. Morrow, and N. S. Kim. 2014. Fair share: Allocation of GPU resources for both performance and fairness. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 440–447.

[3] AMD. 2012. AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE White Paper. *AMD Corporation* (2012).

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

[5] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. 2015. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 25–38.

[6] A. Bakhoda, J. Kim, and T. M. Aamodt. 2010. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 421–432.

[7] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. 2004. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture* 50, 2 (2004), 105 – 128. Special issue on networks on chip.

[8] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.*

[9] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 17–32.

[10] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 681–696.

[11] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H Zhou. 2018. Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[12] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 280–291.

[13] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2010. AéRgia: Exploiting Packet Latency Slack in On-chip Networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 106–116.

[14] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 335–346.

[15] S. Eyerman and L. Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28, 3 (May 2008), 42–53.

[16] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11).* USENIX Association, Berkeley, CA, USA, 323–336.

[17] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012.* 1–10.

[18] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. 2011. Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees. In *2011 38th Annual International Symposium on Computer Architecture (ISCA).*

[19] B. Grot, S. W. Keckler, and O. Mutlu. 2009. Preemptive Virtual Clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[20] H. Jang, J. Kim, P. Gratz, Ki Hwan Yum, and E. J. Kim. 2015. Bandwidth-efficient on-chip interconnect designs for GPGPUs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC).* 1–6.

[21] Qing Jiao, Mian Lu, Huynh Phung Huynh, and Tulika Mitra. 2015. Improving GPGPU Energy-efficiency Through Concurrent Kernel Execution and DVFS. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15).* IEEE Computer Society, Washington, DC, USA, 1–11.

[22] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2014. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU-7).* ACM, New York, NY, USA, Article 1, 8 pages.

[23] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15).* ACM, New York, NY, USA, 12.

[24] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2016. Exploiting Core Criticality for Enhanced GPU Performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS '16).* ACM, New York, NY, 13.

[25] M. Karol, M. Hluchyj, and S. Morgan. 1987. Input Versus Output Queueing on a Space-Division Packet Switch. *IEEE Transactions on Communications* 35, 12 (Dec 1987), 1347–1356.

[26] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47).* IEEE Computer Society, Washington, DC, USA, 114–126.

[27] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *The Sixteenth International Symposium on High-Performance Computer Architecture.*

[28] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43).* IEEE Computer Society, Washington, DC, USA, 65–76.

[29] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. 2012. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. *IEEE Computer Architecture Letters* 11, 2 (July 2012), 33–36.

[30] J. Lee and H. Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *IEEE International Symposium on High-Performance Comp Architecture.* 1–12.

[31] J. W. Lee, M. C. Ng, and K. Asanovic. 2008. Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks. In *2008 International Symposium on Computer Architecture.* 89–100.

[32] Bin Li, Li-Shiuan Peh, Li Zhao, and Ravi Iyer. 2012. Dynamic QoS Management for Chip Multiprocessors. *ACM Trans. Archit. Code Optim.* 9, 3, Article 17 (Oct. 2012), 29 pages.

[33] X. Li and Y. Liang. 2016. Efficient kernel management on GPUs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE).* 85–90.

[34] Yun Liang and Xiuhong Li. 2017. Efficient Kernel Management on GPUs. *ACM Trans. Embed. Comput. Syst.* 16, 4, Article 115 (May 2017), 24 pages.

[35] Zhen Lin, Michael Mantor, and Huiyang Zhou. 2018. GPU Performance vs. Thread-Level Parallelism: Scalability Analysis and a Novel Way to Improve TLP. *ACM Trans. Archit. Code Optim.* 15, 1, Article 15 (March 2018), 21 pages.

[36] Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing thoughput and fairness in SMT processors. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS.* 164–171.

[37] X. Mei and X. Chu. 2016. Dissecting GPU Memory Hierarchy through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2016), 1–1.

[38] O. Mutlu and T. Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007).* 146–160.

[39] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 308–317.

[40] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. 2006. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 208–222.

[41] NVIDIA. 2014. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. (2014).

[42] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 407–418.

[43] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 527–540.

[44] M. K. Qureshi and Y. N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 423–432.

[45] N. Rafique, W. T. Lim, and M. Thottethodi. 2007. Effective Management of DRAM Bandwidth in Multicore Processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 245–258.

[46] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, vLi Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *IMPACT Technical Report* (2012).

[47] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 62–75.

[48] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. 2013. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 639–650.

[49] Y. Ukidave, C. Kalra, D. Kaeli, P. Mistry, and D. Schaa. 2014. Runtime Support for Adaptive Spatial Partitioning and Inter-Kernel Communication on GPUs. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. 168–175.

[50] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *ACM Trans. Archit. Code Optim.* 12, 4, Article 65 (Jan. 2016), 28 pages.

[51] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog. 2018. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[52] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 358–369.

[53] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of Service Support for Fine-Grained Sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 269–281.

[54] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and Exploiting Flexible Task Assignment on GPU Through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 119–130.

[55] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 230–242.

[56] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. 2009. Complexity effective memory access scheduling for many-core accelerator architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[57] J. Zhan, O. KayÄśran, G. H. Loh, C. R. Das, and Y. Xie. 2016. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *49th International Symposium on Microarchitecture (MICRO)*.

[58] J. Zhong and B. He. 2014. Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1522–1532.

[59] Yanqi Zhou and David Wentzlaff. 2016. MITTS: Memory Inter-arrival Time Traffic Shaping. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 532–544.

[60] Amir Kavyan Ziabari, José L. Abellán, Yenai Ma, Ajay Joshi, and David Kaeli. 2015. Asymmetric NoC Architectures for GPU Systems. In *Proceedings of the 9th International Symposium on Networks-on-Chip (NOCS '15)*. ACM, New York, NY.