Improving Inter-kernel Data Reuse With CTA-Page Coordination in GPGPU

Xuanyi Li*, Chen Li*‡, Yang Guo*, Rachata Ausavarungnirun†‡
*National University of Defense Technology, Changsha, China
†King Mongkut's University of Technology North Bangkok, Bangkok, Thailand
*{lixuanyi, lichen, guoyang}@nudt.edu.cn, †r.ausavarungnirun@gmail.com

***Abstract*—Although modern GPUs are equipped with expanding memory, accommodating the entire working set of large-scale workloads can still be a challenge. With the support of unified virtual memory and demand paging, programmers can transparently oversubscribe the main memory. However, this transparent management still comes at a severe performance cost, especially for applications with inter-kernel data sharing. While there have been many efforts to reduce additional data migrations caused by the memory oversubscription, few consider the reuse of shared data during the boundary of adjacent kernels. Due to limited memory capacity, we observe that adjacent kernel often demands shared pages that were evicted by the previous kernel, resulting in a significant number of costly data migrations. In this paper, we propose a CTA-Page collaborative framework, called CPC, that transparently reduces the impact of memory oversubscription using CTA dispatch switching and page replacement switching coordinately to reuse inter-kernel shared data.**

We evaluate CPC with a variety of GPGPU benchmark suites. Experimental results show that the system performance is improved by 65% compared with the state-of-the-art technique for applications with inter-kernel data sharing.

***Index Terms*—memory oversubscription, CTA-Page Coordination, unified memory**

I. Introduction

Due to high computing throughput and increasing programmability, the Graphics Processing Units (GPUs) have been widely used in various fields, including machine learning [1], object detection [2], and image denoising [3] to accelerate large scale parallel computation. However, limited GPU memory capacity cannot satisfy the increasing GPU application working set size, which leads to serious performance slowdown as data needs to be moved back and forth between the CPU memory and the GPU memory.

To manage data between CPU's memory and GPU's memory, programmers have to manually manage the data movement between the CPU memory and the GPU memory, which significantly sacrifices the productivity of programmers. While *Unified Virtual Memory* and *Demand Paging* [4] are introduced to provide transparent memory oversubscription, these techniques still bring high-performance penalties as additional data pages can migrate between the CPU memory and the GPU memory. Therefore, reducing these additional data movements becomes more and more important.

We observe that there are kernels with producer-consumer dependencies and common inputs in many applications according to the analysis across multiple benchmark [5]–[9]. Among these applications, shared data produced or utilized by the CTA of the previous kernel is usually consumed or utilized by the corresponding CTA of the current kernel. For such cases, kernels with data sharing access the same data area at a similar sequence. When the GPU memory is oversubscribed and cannot accommodate the whole working set size of the kernel, old pages will be swapped out to make room for new pages. At the end of each kernel, only the latest accessed pages remain in the GPU memory. When the following kernel launches, the same pages which have been evicted by the previous kernel are accessed again. All these page faults appear at the beginning of the new kernel, causing severe congestion in paging. Although kernels share lots of data with neighbored kernels in these applications, such data sharing can be broken by the memory oversubscription. As shown in Fig.1, data is migrated from the CPU memory to the GPU memory contiguously throughout the whole executing process of the application with inter-kernel data sharing under the oversubscribed GPU memory. Therefore, original data reuse between neighbored kernels no longer exists, leading to a large number of page faults.

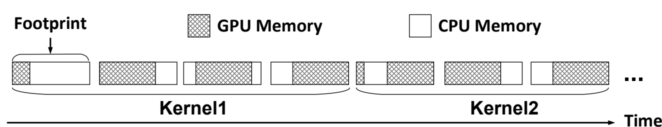


Fig. 1. Paging process of the application with inter-kernel data sharing under the oversubscribed GPU memory.

There are several previous works alleviating the performance loss caused by memory oversubscription, including ETC [10], prefetching pages [11], [12] and batched faulting pages [13], but they can hardly improve the performance for the applications with inter-kernel data sharing. These methods either hide the eviction latency or accelerate the page migration, but they cannot reduce the number of page movements. Therefore, our goal is to explore the opportunities in reusing shared data between the adjacent kernels.

In this paper, we propose a *CTA-Page Collaborative* framework, called CPC, which improves the performance by utilizing the inter-kernel data sharing in an application-transparent

‡ refers to corresponding author.

way. According to the execution of shared-data kernels, CPC coordinates (1) *cooperative thread array (CTA)* dispatch switching (denoted as *DispatchSwitch*) that switches the direction of CTA dispatching sequence and (2) page replacement policy switching (denoted as *ReplacementSwitch*) that switches the page replacement policy to effectively reuse the shared data between kernels. We evaluate CPC with a variety of GPGPU benchmark suites and show that CPC outperforms the state-of-the-art technique by 65%.

This work makes the following contributions:

- This paper provides an in-depth analysis of page access patterns for applications with inter-kernel data sharing. We find that such data reuse is destroyed completely when the memory is oversubscribed since data has to be evicted and paged back and forth among multiple kernels.
- We propose an application-transparent framework, CPC, to reduce the impact caused by the memory oversubscription. Our solution reuses the inter-kernel shared data, which is effective in reducing the data movements between the CPU memory and the GPU memory.
- We find that both *DispatchSwitch* and *ReplacementSwitch* are important to coordinate related computation and data together for reducing the page fault rate at the boundary of neighbored kernels. Overall, CPC outperforms the state-of-the-art technique by 65% with an average 56.7% data reuse rate.

II. Background

This section first introduces the GPU execution model (Section II-A), the architecture of modern GPUs (Section II-B), and the CTA scheduling policy (Section II-C).

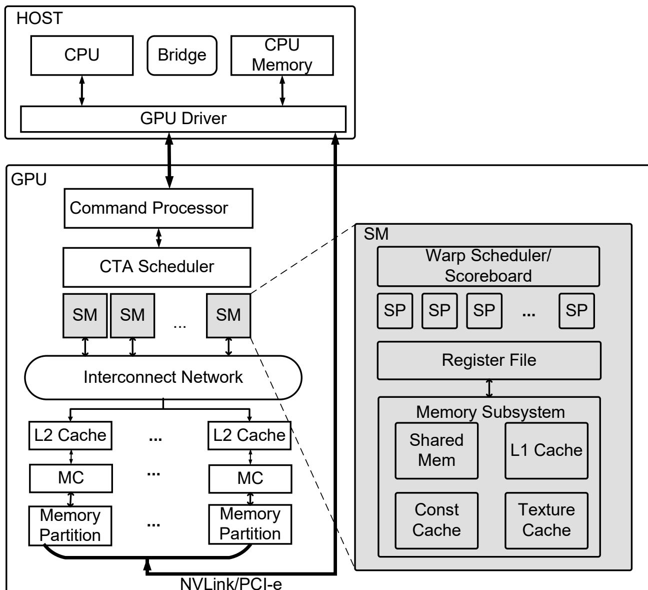


Fig. 2. Baseline GPU Architecture

A. GPU Execution model

GPU applications usually consist of multiple kernels, where kernels without data dependency can run simultaneously. Each GPU kernel consists of groups of threads called *Thread Block (TB)* or *Cooperative Thread Array (CTA)* [1]. A CTA is split into multiple *warps*, where all threads execute in lockstep. Each CTA shares a scratchpad memory called *shared memory*. Though threads within the CTA execute coordinately through various synchronization methods, CTAs can be executed in any order while ensuring the correctness. Therefore, modifying the CTA scheduling policy has no impact on application. Such property is critical for our proposed designs.

With both *Unified Virtual Memory* and *Demand Paging* [4], modern GPUs can transparently manage data, which improves programmability significantly. In this case, data can be accessed with the same virtual address by both CPU and GPU *on demand* instead of all being copied before execution. When the GPU intends to access a physical page absent in the device memory, it generates a page fault and the GPU driver sends data at the page or multi-page granularity from the CPU memory to the GPU memory. This functionality provides the opportunity to oversubscribe the GPU memory. However, this suffers significant performance overhead because the page faults block the execution until the requested data is loaded and such process normally lasts for tens of microseconds [11].

B. GPU Architecture

Fig.2 shows the baseline GPU architecture based on NVIDIA's design. A command processor is used to receive the commands from the CPU and controls the GPU accordingly [14]. The CTA scheduler selects CTAs and dispatches them into the proper SMs based on the status of SMs. A single GPU consists of multiple streaming multiprocessors (SMs) and several shared memory partitions connected via interconnect network. Data transfer between CPU and GPU is through off-chip links such as NVLink and PCI-e. Each SM is composed of multiple streaming processors (SPs) supporting multiple threads executing concurrently. These threads form a warp, which is the basic execution unit. Warps are selected and issued by the warp scheduler from a warp pending pool tracked by the scoreboard, which delivers the high utilization of SMs. Local GPU memory system including various types of L1 caches used in different data spaces for the compute cores. A data access can fetch its data from the caches, the GPU memory via the interconnection network, or from CPU memory through the external link, which can cost hundreds or thousands of cycles [11].

C. CTA Scheduling

Many prior works propose various types of CTA scheduling policy [15]–[19]. The default CTA scheduling policy has been assumed as a simple *round-robin (RR)* scheme [20], [21]. CTAs are selected from the CTA pending pool in ascending

[1]We use NVIDIA's terminology here as our validation platforms are NVIDIA products and the proposed methods are also applicable to other types of GPUs.

order and assigned to each available SMs in multiple rounds by the CTA scheduler (i.e., GigaThread Engine). This process stops until all of the CTAs within the pool have been dispatched or no SMs are available, limited by both resources (e.g., shared memory, registers, etc.) or hardware (e.g., warp slots, etc.). Whenever SMs finish CTAs, new CTAs can be scheduled. In this paper, we also leverage RR as the baseline scheduling policy.

III. Motivation

Efficient GPU memory utilization is critical to avoid the long latency of page faults, especially when the GPU memory is oversubscribed. Fig.3 shows the average performance degradation due to memory oversubscription for applications with inter-kernel data sharing. We make two observations. First, such applications suffer from an average of 198.4% and 183.4% performance loss when 75% of the total memory footprint fit in the GPU memory and when the oversubscription ratio varies from 50% to 95%, respectively. Second, the performance impact of memory oversubscription to such applications is insensitive to the degree of oversubscription, indicating that the performance of such applications degrades drastically once the memory is oversubscribed.

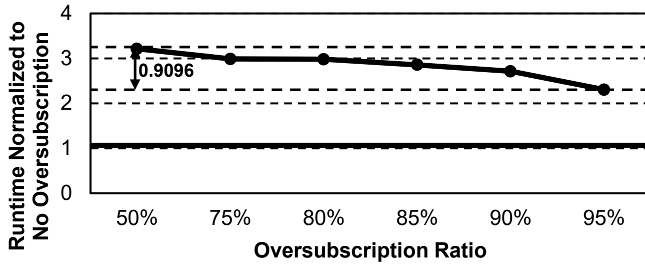


Fig. 3. Impact of memory oversubscription to the performance of applications with inter-kernel data sharing. The performance drop is only 0.9096 when the oversubscription ratio decreases from 95% to 50%.)

Fig.4 shows both the page access pattern and page fault rate of FFT when the GPU memory can only fit 75% of its footprint. We find that each kernel has a similar access pattern and sequence. As shown in the Fig.4 with blue circles, there are amounts of page faults at the boundary of kernels. When a new kernel is launched, all new threads are dispatched to SMs in the GPU and start accessing those pages at the same time. However, all these pages have been evicted due to memory oversubscription at the end of the last kernel shown in Fig.4. Moreover, since the kernel is newly launched, all dispatched threads are seeking new pages together. It causes these large amounts of page faults. In order to reduce the number of page migrations under the oversubscribed GPU memory, previous works [10]–[13] have been proposed to reduce the migration overhead by prefetching [11], [12], overlapping eviction latency with paging [10] and batched faulting pages together [13]. However, these techniques provide little performance benefit for inter-kernel data sharing applications. Prefetching would cause severe thrashing for such applications because it may evict used pages proactively. Proactive eviction can hide some eviction latency, but it cannot help with reducing such page faults. Batched faulting pages is effective in amortizing the page fault overhead, but it is useless for regular applications with streaming access pattern.

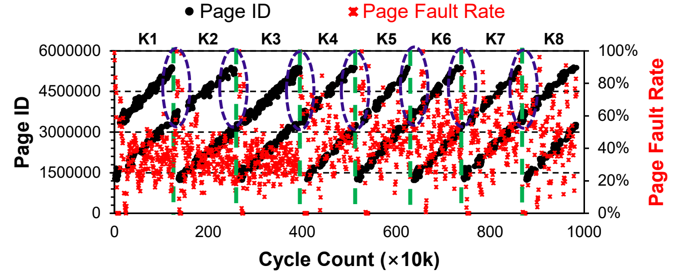
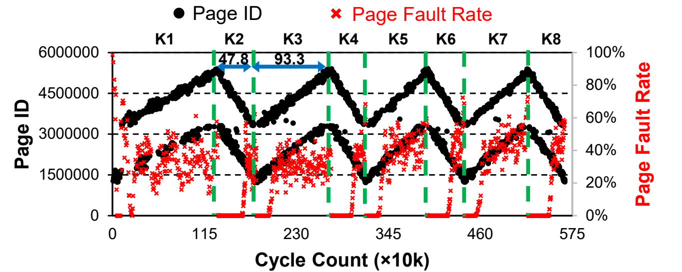
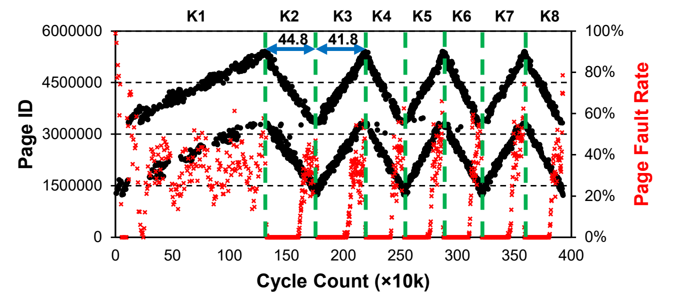


Fig. 4. Page access pattern and page fault rate of FFT when 75% of the footprint can be fit into the memory (kernels within FFT access the same pages in the same sequence as shown by the black dots; dashed line represents the end of each kernel; page fault rate is high at the boundary of neighbored kernels as shown in the circled regions).

In summary, we make two observations for applications with inter-kernel data sharing. First, once memory is oversubscribed, shared data reusing is broken at the boundary of kernels and the performance drops sharply due to thrashing. Second, previous works have not considered data reuse issues for inter-kernel data sharing under memory oversubscription. Therefore, the goal of this paper is to alleviate the overhead of memory oversubscription through reusing the shared data.



(a) *DispatchSwitch* is applied (page fault rate is low at the boundary of neighbored kernels but the odd-round kernels cost more time than the even-round kernels)



(b) *CPC* is applied (execution time of the odd-round kernels is reduced a lot)

Fig. 5. Page access pattern and page fault rate of FFT using *DispatchSwitch* and CPC when 75% of the footprint can be fit into the GPU memory (kernels within FFT access the same pages at the same sequence as shown by the black dots; dashed line represents the end of each kernel).

IV. CPC DESIGN

Based on the analysis in Section III, we observe that coordinating CTAs and the remaining data in the memory is a key solution to reduce the page fault rate at the boundary of kernels. This observation guides the design of the *CTA-Page collaborative* framework, CPC. The key principle of CPC is to 1) dispatch related CTAs first to reuse the data remained in the memory through CTA dispatch switching (Section IV-A) and 2) replace useless pages to cooperate with the corresponding CTA dispatching strategy through page replacement switching (Section IV-B).

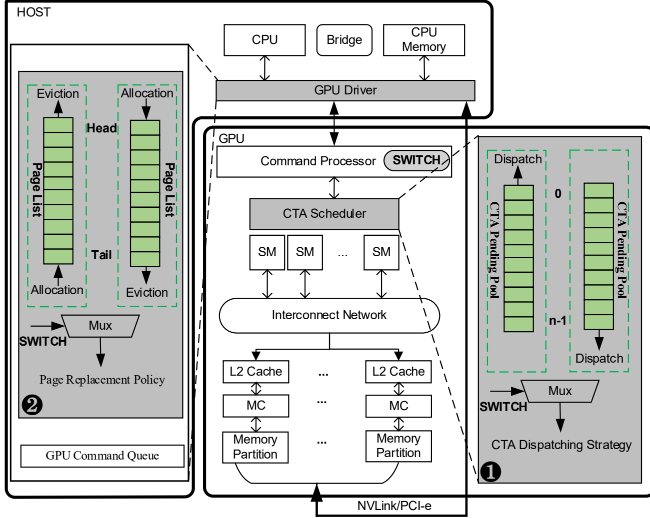


Fig. 6. Architecture of CPC (❶: CTA Dispatch Switching; ❷: Page Replacement Switching)

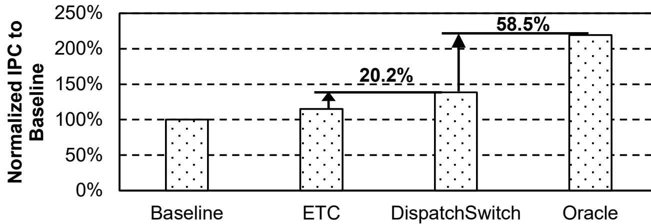


Fig. 7. Performance of *DispatchSwitch*, ETC, Oracle normalized to the baseline.

A. CTA Dispatch Switching

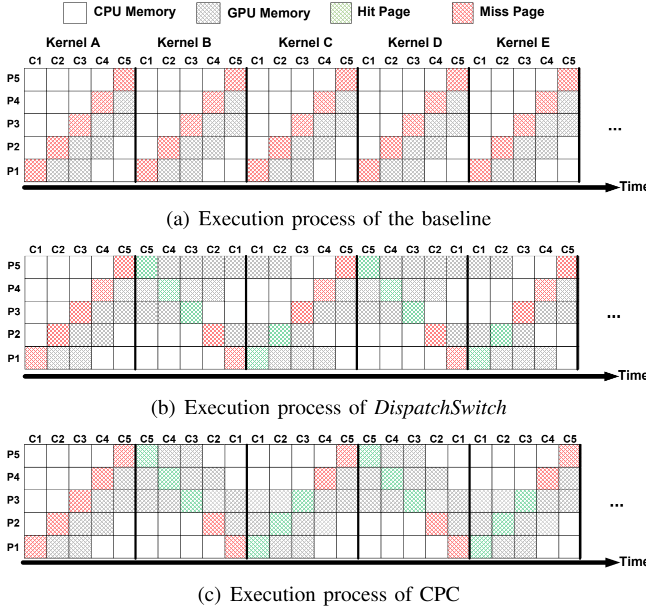
However, how to find the CTAs related to the pages in the memory becomes our first design challenge. Observed from applications with inter-kernel sharing, we find that most of these applications have similar access patterns among different kernels, which is also shown in the Fig.4. Within each kernel, such page access behavior is strongly related to the CTA scheduling scheme, since each thread typically uses thread ID and associated CTA ID to identify the data that it operates on.

To this end, we propose a CTA dispatch switching mechanism, called *DispatchSwitch*. It simply switches the dispatching order of CTAs whenever a new kernel is launched to utilize the remaining data in the GPU memory. As shown in ❶ of Fig.6, the switch flag denoted as SWITCH is reset when a new context is created and is reversed by the command processor whenever a new kernel is launched. For each kernel, the CTA scheduler selects corresponding CTA dispatching strategy based on the SWITCH. When SWITCH equals 0, CTAs of the current kernel are dispatched in ascending order otherwise in descending order. Thereby remaining data in the GPU memory is utilized effectively by the following kernel.

To show the effectiveness of this mechanism, we compare it against the state-of-the-art framework, called ETC, which focuses on reducing the impact of memory oversubscription [10]. We also introduce an oracle design in which all shared data between neighbored kernels within the GPU memory is reused. Fig.7 presents the average performance of *DispatchSwitch*, ETC and Oracle normalized to the baseline using the default configurations as shown in Table I. Page access pattern and page fault rate of FFT using *DispatchSwitch* are shown in Fig.5(a). We make three observations from these figures. First, compared to ETC [10], *DispatchSwitch* improves the performance by 20.2% on average in line with the page fault reduction at the boundary of kernels. It means that *DispatchSwitch* is effective in reusing shared data. Second, *DispatchSwitch* performs 58.5% worse than Oracle, indicating there is an opportunity to further reuse the shared data. Third, an interesting finding is that the execution time of the even-round kernels is shorter than that of the odd-round kernels as shown in the Fig.5(a). It leads us to further dig into the difference between the odd and even rounds of kernels.

We create and analyze the execution process of a simple workload with multiple kernels which access the same data as shown in Fig.8. In order to simplify our analysis, we make several assumptions as shown in Fig.8. We first analyze the execution process of the baseline as shown in Fig.8(a). Kernels are launched sequentially and CTAs are also dispatched in ascending order. At first, requested pages are loaded into the empty GPU memory. Due to the limited memory capacity and age-based LRU page replacement policy (first loaded, first evicted) [22], the earlier loaded pages are evicted first to make room for new pages. Therefore, throughout the whole execution process, no data requests hit in the GPU memory, leading to performance degradation consistent with Fig.3.

When *DispatchSwitch* is applied, the execution process is shown in Fig.8(b). Different from the baseline, the CTA dispatching sequences are switched every kernel. We make two observations. First, there are fewer page faults at the boundary of kernels compared to the baseline consistent with the page faults reduction at the boundary (Fig.5(a)). Second, for odd-round kernels, the remaining data is not fully reused accounting for the performance gap with the oracle case. In summary, switching the CTA dispatching sequence can provide a significant performance benefit, but it still cannot fully utilize the remaining data in the GPU memory. Coordinating the page replacement policy with *DispatchSwitch* is key to bridge such a performance gap.



(a) Execution process of the baseline

(b) Execution process of *DispatchSwitch*

(c) Execution process of CPC

Fig. 8. Execution process of the microbenchmark (To simplify the analysis, we make the following assumptions. 1) Only one CTA can run at the same time. 2) Five kernels (A, B, C, D, E) access the same data and each kernel consists of five CTAs (C1-C5) accessing only one data page (P1-P5, respectively). 3) the GPU memory capacity is 3 pages.).

B. Page Replacement Policy Switching

As discussed in Section IV-A, odd-round kernels underutilize the shared data, as *DispatchSwitch* may mismatch with the age-based LRU policy. The right part of ❷ in Fig.6 shows how age-based LRU policy works. To provide the support of demand paging, the GPU driver within the host maintains a single list on recording migration order of pages from the CPU memory to the GPU memory. Although it cannot show the access order of pages, it maintains the migration order with low overhead when compared with ideal LRU [23]. Since our CTA scheduler switches the execution order for each kernel, traditional age-based LRU policy cannot cooperate with CTA scheduling mechanism anymore. As shown in Fig.8(b), Page 3 (P3) is evicted instead of Page 5 (P5) at the C2 of Kernel B according to the age-based LRU policy. However, P5 is the old page in Kernel B, which is preferred to be evicted.

To address this issue, we propose a page replacement policy switching mechanism, called *ReplacementSwitch*, as shown in ❷ of Fig.6. *ReplacementSwitch* introduces a new page replacement policy, called reverse age-based LRU policy shown in the left part of ❷ in Fig.6. Different from age-based LRU, the direction of both page eviction and page allocation is switched. Similar to *DispatchSwitch*, the GPU driver selects the corresponding page replacement policy based on SWITCH. When SWITCH equals 0, the GPU driver applies age-based LRU as page replacement policy, otherwise reverse age-based LRU is applied. At this time, CTA dispatching strategy and page replacement policy change simultaneously to cooperate for effective data reuse whenever a new kernel is launched.

Fig.8(c) shows the execution process of the microbenchmark introduced in the previous section when both *DispatchSwitch* and *ReplacementSwitch* (this combination is denoted as CPC), are applied. As expected, Page 5 (P5) is evicted at the C2 of Kernel B according to the reverse age-based LRU. Therefore, all the remaining pages within the GPU memory are utilized by Kernel C with no page faults at the boundary of kernels. Fig.5(b) shows the page access pattern and page fault rate of FFT using CPC. We make two observations. First, the performance of odd-round kernels is improved significantly, leading to better overall performance than the case only using *DispatchSwitch* shown in Fig.5(a). Second, more remaining pages are reused resulting in fewer page faults at the boundary of kernels. We conclude that CPC is effective at reducing page faults at the boundary of kernels via reusing the shared data.

V. Methodology

We extend the GPGPU-Sim v4.0.0 [24] in our experiments. The configurations of the GPU system including both cores and memory are shown in TABLE I.

TABLE I
CONFIGURATIONS OF THE SIMULATED SYSTEM

GPU Core Configurations	
GPU Arch	NVIDIA RTX-2060 Turing-like
GPU Cores	30 CUs @ 1.4GHz, 32 threads per warp, 4 GTO scheduler
Private L1 Caches	64KB/CU, fully associative, LRU
Shared L2 Caches	Memory Side 256KB/DRAM Channel, 16-way associative, LRU
Memory System Configurations	
DRAM	GDDR6, 12-channel, FR-FCFS scheuler, RCD=RP=20, RC=62, CL=WR=20
Unified Memory	4KB page size, 20μs page fault latency, 16 GB/s PCIe bandwidth

Demand Paging and Oversubscription. We faithfully model demand paging between the CPU memory and the GPU memory. Whenever data absent in the GPU memory is requested, page faults are generated and the GPU driver resident in the host serves this exception by sending the required pages to the GPU. If the GPU memory is full, pages in the memory should be evicted based on the page replacement policy via the GPU driver. To experiment under different memory oversubscription degrees, the capacity of the GPU memory is configured to fractions (from 75% to 95%) of each workload's footprint.
Workloads. We select 12 applications from CUDA SDK [9], Rodinia [5], Parboil [6], Ispass [7] and Polybench [8] benchmark suites. The footprint of these workloads varies from 1MB to 96MB with an average of 18.5MB. Limited simulating speed hinders us from emulating a larger footprint.
Evaluation Metrics. We use normalized IPC to evaluate the performance of proposed mechanisms. To report the data reuse rate, we introduce a formula defined as $\frac{Access_{shared}}{Access_{Total}}$, where $Access_{shared}$ is the number of accesses requesting the inter-kernel shared pages and $Access_{Total}$ is the number of total

accesses. Page fault rate is used to illustrate the stability of the memory system when the GPU memory is oversubscribed.

VI. Evaluation

We evaluate both *DispatchSwitch* (denoted as CPC-Disp) and CPC via comparing them against four designs: 1) baseline 2) ETC [10] 3) Oracle 4) an ideal baseline with unlimited GPU memory. Since above designs are all orthogonal to ETC, we also evaluate these designs when equipped with ETC. The configurations of these designs are shown in TABLE II.

A. Performance

Fig.9 shows the average IPC normalized to the baseline with our proposed mechanisms. From the figure, we make four observations. First, the state-of-the-art mechanism ETC shows modest benefit with an average of 15% performance improvement. As Section III states, the performance of applications with inter-kernel data sharing degrades significantly once the GPU memory is oversubscribed. Therefore memory compression hardly works. Second, CPC outperforms the baseline and ETC by 90% and 65% respectively and obtains a similar performance benefit to Oracle, indicating that CPC can reuse the inter-kernel shared data effectively. Third, CPC outperforms DispatchSwitch by 37.3%, illustrating that both *DispatchSwitch* and *ReplacementSwitch* play important roles. Last, all methods equipped with ETC improve the performance further, showing that CPC and ETC are orthogonal with each other which can work together.

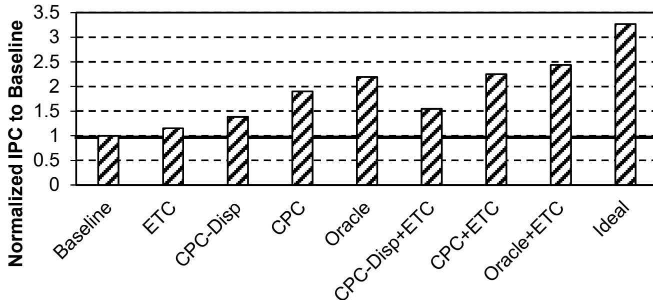


Fig. 9. Overall performance normalized to the baseline.

Individual Workload Performance. Fig.10 shows the performance of CPC and the other techniques normalized to the baseline across multiple workloads. We make three observations. First, performance across all of the workloads has been improved with CPC compared to the baseline, especially for SRADV2 (330%) and SPMV (292%). Second, for most of the workloads, CPC outperforms ETC and *DispatchSwitch* and obtains a similar performance to the oracle case. Third, there is little performance improvement for some applications like BFS, 3DCONV and FWT when CPC is applied because little shared data between neighbored kernels leaves little room for improvement. We conclude that CPC is effective at reducing the performance impact of memory oversubscription for applications with inter-kernel data reuse.

B. Page Fault Rate

We next investigate where such significant performance improvement originates from through quantifying the page fault rate as shown in Fig.11, from which we make three observations. First, in line with the performance results described in the previous section, we observe that CPC reduces the page fault rate across all the workloads and there is an average reduction of 87.3% and up to 266.3% (SRADV2). Second, for most of the applications, there is little page fault rate reduction when ETC is applied since such applications are insensitive to the memory size under the oversubscribed memory as stated in Section III. Third, the page fault rate of BFS even under the baseline is low, indicating that there remains little performance improvement space, consistent with the results just described.

C. Data Reuse Rate

We further investigate where the page fault reduction is coming from. We quantify the data reuse rate when 75% of the footprint fits in the GPU memory (used in ETC [10]) as shown in Fig.12. From the figure, We make three observations. First, little shared data is reused for the baseline for most of the applications as shown in Fig.8(a). Second, CPC provides an average of 56.7% data reuse rate and up to 74.9% (SRADV2), which is consistent with the page fault rate reduction stated in the last section because more data to be reused leads to fewer page faults. Third, when equipped with ETC, CPC can deliver an average of 75.4% data reuse rate and up to 90% (SRADV2), indicating that almost all the shared data is reused.

D. Sensitivity Analysis

In this section, we measure the sensitivity of CPC's effectiveness to the degree of memory oversubscription, the page size and the page fault handling latency.

Memory Oversubscription Ratio. From Fig.3, we find that different memory oversubscription ratios have similar impacts on the performance. Fig.13(a) shows the performance improvement of CPC when the memory oversubscription ratio varies from 75% (i.e., only 75% of each workload's working set can fit in the GPU memory) to 100% (i.e., there is no memory oversubscription). We observe that the performance benefit CPC brings increases as the GPU memory becomes larger because more data remained in the GPU memory can be reused. We conclude that CPC can provide amounts of performance benefit under the oversubscribed GPU memory.

Page Size. Page size plays a critical role in the virtual memory traffic observed on both CPUs and GPUs. Fig.13(b) shows the average performance benefit of CPC when page size changes from 1KB to 32KB normalized to the baseline. We observe that the performance improvement decreases as the page size becomes larger. Applications access less pages as the page becomes larger leading to less page faults and less page migrations when the GPU memory is oversubscribed. Though there remains little improvement space to reduce the page faults for applications with larger pages, results shows

TABLE II
CONFIGURATIONS OF DIFFERENT DESIGNS

Design	Memory Oversubscription	CTA Scheduling	Page Replacement	Memory Compression	Proactive Eviction
Baseline	✓	RR in ascending order	Age-based LRU	✗	✗
ETC	✓	RR in ascending order	Age-based LRU	✓	✓
CPC-Disp	✓	*DispatchSwitch*	Age-based LRU	✗	✗
CPC	✓	*DispatchSwitch*	*ReplacementSwitch*	✗	✗
Oracle	✓	RR in ascending order	Data accessed before hit	✗	✗
CPC-Disp + ETC	✓	*DispatchSwitch*	Age-based LRU	✓	✓
CPC + ETC	✓	*DispatchSwitch*	*ReplacementSwitch*	✓	✓
Oracle + ETC	✓	RR in ascending order	Data accessed before hit	✓	✓
Ideal	✗	RR in ascending order	Age-based LRU	✗	✗

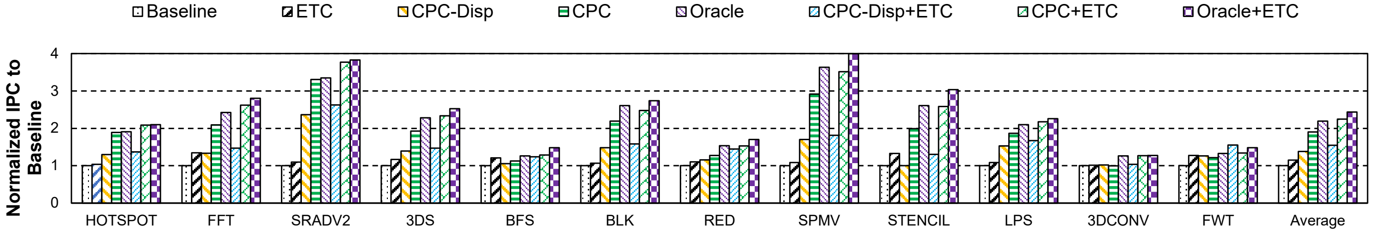


Fig. 10. Overall performance normalized to the baseline for different applications.

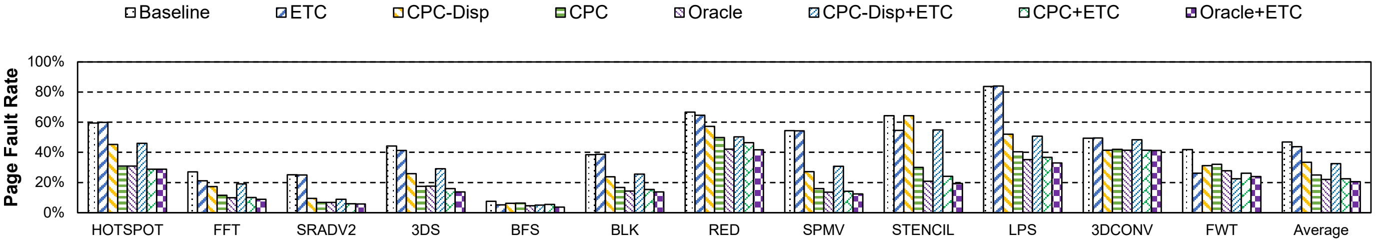


Fig. 11. Page fault rate for different applications.

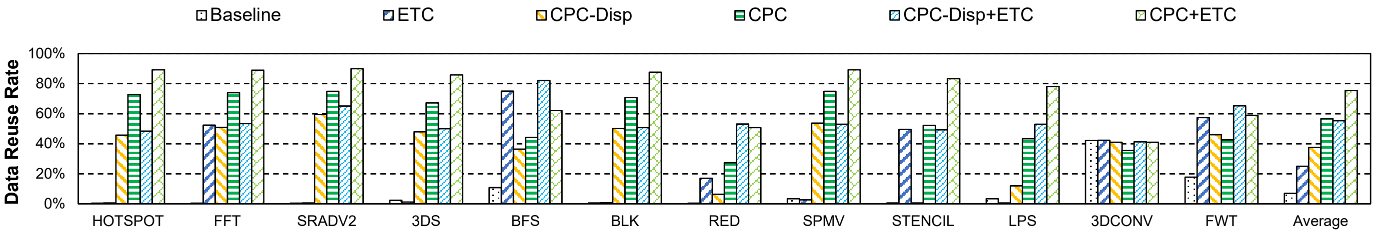


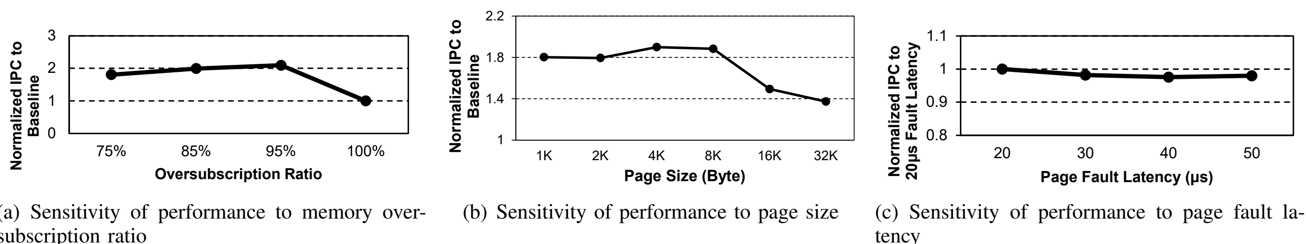
Fig. 12. Data reuse rate for different applications.

our mechanism is still successful in alleviating the impact of memory oversubscription.

Page Fault Latency. The page fault latency is critical to the performance especially when the memory is oversubscribed. To understand the impact of such latency, we perform sensitiveness studies. Fig.13(c) presents the average performance improvement with CPC when the page fault latency is varied from 20μs to 50 μs normalized to the configuration with 20μs fault latency. We observe that CPC can always brings significant performance improvement with any page fault latency because it focuses on reducing page faults and is insensitive to the page fault latency, which shows the effectiveness of our approach in reducing the impact of memory oversubscription.

E. Hardware Overhead

We analyze the hardware overhead to support each component of CPC. To implement *DispatchSwitch*, the CTA scheduler needs extending with corresponding control logic with almost no hardware overhead to support two CTA dispatching strategies. *ReplacementSwitch* is implemented in the GPU driver with no hardware overhead. Besides, a single bit is added to the command processor to indicate which strategy to



(a) Sensitivity of performance to memory oversubscription ratio

(b) Sensitivity of performance to page size

(c) Sensitivity of performance to page fault latency

Fig. 13. Sensitivity studies for CPC

select for both *DispatchSwitch* and emphReplacementSwitch. Overall, hardware overheads for our design are negligible.

VII. Related work

To the best of our knowledge, this paper is the first to reuse the inter-kernel shared data to improve the performance under the oversubscribed memory. We survey related works in GPU memory oversubscription, CTA scheduling, exploiting data locality and replacement policies.

GPU Memory Oversubscription. Kim *et al.* [13] show the high cost of page faults and propose a scheme to increase the number of page faults handled together to amortize the fault handling latency across multiple pages. Ganguly *et al.* [25] propose a programmer-agnostic scheme to decide whether to copy the data from the CPU memory to the GPU memory based on the classification of applications. García-Flores [26] *et al.* propose a fine-grained data transferring scheme to make the most use of limited bandwidth, reduce the transmission delay and alleviate memory contention. Zheng [11] *et al.* propose a scheme to hide the long latency by viewing the page fault as the store operation. The above methods all focus on how to reduce the impact of the page fault and none consider the data reuse among kernels. These approaches are orthogonal to our work.

CTA Scheduling. Several works explore inter-CTA locality via scheduling the CTAs properly [15]–[19]. Li *et al.* [15] propose a software scheme that relocates CTAs which share data to the same SM to utilize the inter-CTA locality based on the inherent scheduling policy of real GPU hardware. Wang *et al.* [16] deliver consecutive CTAs to the same cluster to coalesce requests to the same data and reduce the redundant traffic. Lee *et al.* [18] dispatch a group of two adjacent CTAs to the same SM to improve the L1 performance. Chen *et al.* [19] modify the CTA scheduling policy to improve the inter-CTA locality. None of these works explore the inter-kernel locality. Huzaifa *et al.* [17] deploy several CTA scheduling policies to utilize inter-kernel locality within L1 cache, but they do not consider how to utilize such locality within the GPU memory to reduce the performance impact of memory oversubscription.

Exploiting Data Locality. Ang *et al.* [15] and Wang *et al.* [16] schedule a group of CTAs each time to utilize inter-CTA locality. Tabbakh *et al.* [27] propose a share-aware cache scheme that maintains private data in the L1 cache and shared data in the L2 cache. Vijaykumar *et al.* [28] propose a hardware and software cooperative scheme to utilize the data locality marked explicitly. These techniques targets intra-kernel locality but are orthogonal to our designs since we focus on inter-kernel locality. Huzaifa *et al.* [17] utilize inter-kernel data locality within the L1 cache through several simple schemes. Due to the small L1 cache size, it performs badly for kernels with a large working set. Our research coordinates both the CTA scheduling policy and the replacement policy to exploit inter-kernel locality.

Replacement Policies. Lots of works research in replacement policy for cache and memory in both CPU and GPU [29], [30]. LRU (*Least Recent Used*) [23] is widely used and performs well in the cache system. While for global memory, it is unrealistic to implement LRU due to its expensive cost. Some variants like NRU (*Not Recently Used*) [31] and CLOCK (*Second Chance Replacement*) [32] are commonly used. These policies perform well for applications with strong temporal locality while performing badly for those with thrashing access patterns. LFU (*Least Frequently Used*) [33] and DIP (Dynamic Insertion Policy) [34] target workloads with large working set and thrashing access patterns. Q. Yu *et al.* [35] propose a hierarchical page eviction scheme which selects proper page eviction policy for each application detected at runtime. To protect intra-warp locality, Koo *el al.* [36] propose access pattern-aware cache management scheme, which bypasses streaming accesses and protects frequently used data from being evicted. These policies except LRU do not match with our design but can be employed to the cache system.

VIII. Conclusion

Applications with inter-kernel data sharing suffer from great performance loss under memory oversubscription. To alleviate such performance degradation, we propose CPC, an application-transparent framework that coordinates CTA dispatching strategy and page replacement policy to effectively reuse the shared data for reducing the performance impact of memory oversubscription. CPC reduces the page fault rate by an average of 87.3% compared to the baseline, leading to an average 90% and 65% performance improvement than the baseline and the state-of-the-art memory management framework ETC, respectively. We conclude that CPC is an effective low-cost framework to exploit inter-kernel data sharing.

Acknowledgment

We thank the anonymous reviewers from ICCAD 2021. This work is supported by Research Project of NUDT ZK20-04.

REFERENCES

[1] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, X. Wang, and Ieee, *Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective*, ser. International Symposium on High-Performance Computer Architecture-Proceedings. Institute of Electrical and Electronics Engineers, 2018, pp. 620–629.

[2] B. Tekin, S. N. Sinha, P. Fua, and Ieee, *Real-Time Seamless Single Shot 6D Object Pose Prediction*, ser. IEEE Conference on Computer Vision and Pattern Recognition. Institute of Electrical and Electronics Engineers, 2018, pp. 292–301.

[3] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising," *Ieee Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017.

[4] T. C. Schroeder, "Peer-to-peer & unified virtual addressing," in *GPU Technology Conference, NVIDIA*, 2011.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Institute of Electrical and Electronics Engineers, 2009, pp. 44–54.

[6] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[7] M. Giles, "Jacobi iteration for a laplace discretisation on a 3d structured grid," 2008.

[8] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*. Ieee, 2012, pp. 1–10.

[9] N. Corp., "Nvidia cuda sdk code samples," % url-http://developer.nvidia.com/.

[10] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, J. Yang, and Acm, *A Framework for Memory Oversubscription Management in Graphics Processing Units*, ser. Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, 2019.

[11] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Institute of Electrical and Electronics Engineers, 2016, Conference Proceedings, pp. 345–357.

[12] Q. Yu, B. Childers, L. Huang, C. Qian, H. Guo, and Z. Wang, "Coordinated page prefetch and eviction for memory oversubscription management in gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Institute of Electrical and Electronics Engineers, 2020, pp. 472–482.

[13] H. Kim, J. Sim, P. Gera, R. Hadidi, H. Kim, and Acm, *Batch-Aware Unified Memory Management in GPUs for Irregular Workloads*, ser. Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, 2020.

[14] J. J. K. Park, Y. Park, and S. Mahlke, "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*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 527–540.

[15] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," *Operating Systems Review*, vol. 51, no. 2, pp. 297–311, 2017.

[16] L. Wang, X. Zhao, D. Kaeli, Z. Wang, and L. Eeckhout, "Intra-cluster coalescing and distributed-block scheduling to reduce gpu noc pressure," *Ieee Transactions on Computers*, vol. 68, no. 7, pp. 1064–1076, 2019.

[17] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, "Inter-kernel reuse-aware thread block scheduling," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, Aug. 2020.

[18] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Institute of Electrical and Electronics Engineers, 2014, pp. 260–271.

[19] L. Chen, H. Cheng, P. Wang, and C. Yang, "Improving gpgpu performance via cache locality aware thread block scheduling," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 127–131, 2017.

[20] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 395–406, 2013.

[21] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Laperm: Locality aware scheduler for dynamic parallelism on gpus," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 583–595.

[22] M. Knap and P. Czarnul, "Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus," *The Journal of Supercomputing*, vol. 75, no. 11, pp. 7625–7645, 2019.

[23] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[24] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[25] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 451–461.

[26] V. Garcia-Flores, E. Ayguadé, and A. J. Peña, "Efficient data sharing on heterogeneous systems," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 121–130.

[27] A. Tabbakh, M. Annavaram, X. Qian, and Ieee, *Power Efficient Sharing-Aware GPU Data Management*, ser. International Parallel and Distributed Processing Symposium IPDPS. Institute of Electrical and Electronics Engineers, 2017, pp. 698–707.

[28] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, O. Mutlu, and Ieee, *The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs*, ser. Conference Proceedings Annual International Symposium on Computer Architecture. Institute of Electrical and Electronics Engineers, 2018, pp. 829–842.

[29] X. Zhao, M. Jahre, and L. Eeckhout, "Selective replication in memory-side gpu caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 967–980.

[30] X. Zhao, A. Adileh, Z. Yu, Z. Wang, A. Jaleel, and L. Eeckhout, "Adaptive memory-side last-level gpu caching," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 411–423.

[31] I. Sun Microsystems, "Ultrasparc t2 supplement to the ultrasparc architecture 2007."

[32] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement." in *FAST*, vol. 4, 2004, pp. 187–200.

[33] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Computer Architecture Letters*, vol. 50, no. 12, pp. 1352–1361, 2001.

[34] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.

[35] Q. Yu, B. Childers, L. Huang, C. Qian, Z. Wang, and Ieee, *Hierarchical Page Eviction Policy for Unified Memory in GPUs*, ser. 2019 Ieee International Symposium on Performance Analysis of Systems and Software. Institute of Electrical and Electronics Engineers, 2019.

[36] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in gpu," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, Conference Proceedings, pp. 307–319.