A Framework for Memory Oversubscription Management in Graphics Processing Units

Chen Li[1,2] Rachata Ausavarungnirun[3,6] Christopher J. Rossbach[4,5]
Youtao Zhang[2] Onur Mutlu[7,3] Yang Guo[1] Jun Yang[2]
[1]National University of Defense Technology [2]University of Pittsburgh
[3]Carnegie Mellon University [4]University of Texas at Austin [5]VMware Research
[6]King Mongkut's University of Technology North Bangkok [7]ETH Zürich

Abstract

Modern discrete GPUs support unified memory and demand paging. Automatic management of data movement between CPU memory and GPU memory dramatically reduces developer effort. However, when application working sets exceed physical memory capacity, the resulting data movement can cause great performance loss.

This paper proposes a memory management framework, called ETC, that transparently improves GPU performance under memory oversubscription using new techniques to overlap eviction latency of GPU pages, reduce thrashing cost, and increase effective memory capacity. Eviction latency can be hidden by eagerly creating space for demand-paged data with *proactive eviction (E)*. Thrashing costs can be ameliorated with *memory-aware throttling (T)*, which dynamically reduces the GPU parallelism when page fault frequencies become high. *Capacity compression (C)* can enable larger working sets without increasing physical memory capacity. No single technique fits all workloads, and, thus, ETC integrates proactive eviction, memory-aware throttling and capacity compression into a principled framework that dynamically selects the most effective combination of techniques, transparently to the running software. To this end, ETC categorizes applications into three categories: *regular applications without data sharing* across kernels, *regular applications with data sharing* across kernels, and *irregular applications*. Our evaluation shows that ETC fully mitigates the oversubscription overhead for regular applications without data sharing and delivers performance similar to the *ideal* unlimited GPU memory baseline. We also show that ETC outperforms the state-of-the-art baseline by 60.4% and 270% for *regular applications with data sharing* and *irregular applications*, respectively.

CCS Concepts • **Computer systems organization Single instruction, multiple data**; • **Software and its engineering** *Virtual memory*.

Keywords graphics processing units; GPGPU applications; virtual memory management; oversubscription

ACM Reference Format:
C. Li et al. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, April 13–17, 2019 (ASPLOS'19),* 15 pages.

ASPLOS'19, April 13–17, 2019, Providence, RI, USA

ACM ISBN ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304044

1 Introduction

Increased compute density and improved programmability [1, 66] have made Graphics Processing Units (GPUs) a platform of choice for high performance applications. However, maximizing application performance still requires arduous hand tuning of applications [97] to fit GPU architectures and physical memory capacity. As General Purpose GPU (GPGPU) application working set sizes increase [36, 50, 59, 78], limited memory capacity becomes a first order design and performance bottleneck [78, 79, 102].

Improved memory virtualization support has recently emerged to allow GPGPU applications to easily extend their working set beyond the limit of a GPU's physical memory [7, 9, 10, 13, 19, 37, 76, 77, 96]. Modern GPUs [56, 67, 68] are now equipped with *unified memory* and *demand paging*. These features free developers from manually managing data movement between the CPU and GPU memory. However, when a GPU kernel working set exceeds the GPU physical memory capacity, i.e., when the GPU memory is *oversubscribed*, data must be swapped in and out of GPU memory on demand. Our measurements on a real GPU system (§2.1) show that real GPGPU applications experience crippling slowdowns and sometimes crash when a fraction of their allocated space does *not* fit in GPU memory.

Some of the performance loss from memory oversubscription can be reduced via more programming effort [33, 83–86]. For example, programmers can duplicate read-only data in both CPU and GPU memory without the need for eviction from GPU memory to CPU memory when the data is no

longer used. Programmers can also overlap prefetch requests with the eviction requests to hide eviction latency. However, solving the memory oversubscription problem with software modifications has significant drawbacks. First, it forces programmers to distinguish between read and write data explicitly. Second, programmers must understand and leverage data locality occurring across thousands of concurrent hardware threads to explicitly map pages to the CPU memory or the GPU memory. Third, programmers need to manually manage data migration between the CPU and the GPU. These limitations are exacerbated in a cloud environment, where VMs may share a GPU and have no visibility into the working set sizes of other tenants' applications. Application-transparent mechanisms that can maintain a good level of performance in the presence of memory oversubscription are urgently needed.

We observe two key properties of contemporary GPGPU applications that can lead to better management of oversubscribed memory. First, performance degradation due to oversubscription varies across applications due to applications' different memory access behavior. We broadly categorize applications into *regular* and *irregular* applications, according to the predictability of their GPU memory page accesses. Second, the dominant source of memory oversubscription overhead differs by application category. Thrashing, which occurs when pages are demand-migrated between the host and the GPU memory repeatedly, dominates the performance overhead for oversubscribed *irregular* applications, while long-latency evictions dominate the overhead for *regular* applications. We also find that data sharing between different GPU kernels from the same application further impacts how the GPU should manage the oversubscribed memory.

Building on our key observations, we propose a memory oversubscription management framework, called **E**viction-**T**hrottling-**C**ompression (ETC), to reduce GPU memory oversubscription overheads in an application-transparent manner. ETC first efficiently and automatically classifies applications into three categories: *regular applications with no data sharing*, *regular applications with data sharing* or *irregular applications*. Second, ETC selects an effective combination of mechanisms for each running application to mitigate the memory oversubscription overhead based on that classification. ETC integrates a number of components that work harmoniously to hide or reduce performance overheads of memory oversubscription. ETC comprises (1) a *classifier* that detects each application's type based on the measured memory coalescing factors; (2) a *policy engine* that selects and applies amelioration techniques based on application type; (3) a *proactive eviction* technique for regular applications, which opportunistically creates capacity for demand-fetched data in advance; (4) a *memory-aware throttling* technique for irregular applications which reduces effective working set sizes by reducing the application's thread-level parallelism; and (5) a main memory compression engine, which transparently increases effective physical memory capacity for GPGPU applications.

We implement ETC as a hardware/software cooperative runtime. We evaluate ETC using 15 applications from a variety of GPGPU benchmark suites. Our evaluations show that ETC is effective at reducing oversubscription overheads. For *regular applications with no data sharing*, ETC eliminates the overhead of memory oversubscription and delivers performance similar to the *ideal* unlimited memory baseline. For *regular applications with data sharing* and *irregular applications*, ETC outperforms the state-of-the-art baseline by 60.4% and 270%, respectively.

This paper makes the following contributions:

- To our knowledge, this is the first paper to 1) provide an in-depth analysis of the performance overhead due to memory oversubscription in GPUs and 2) identify sources of performance loss due to memory oversubscription for different types of GPGPU applications.
- We propose a new hardware/software cooperative solution that significantly reduces the impact of memory oversubscription in GPUs. Our solution, ETC, requires no programmer effort and no modifications to application code.
- We develop three memory oversubscription mitigation techniques as part of ETC. We find that no single mitigation technique fits all types of workload. To this end, ETC classifies applications based on the regularity of their memory accesses and uses the most effective combination of techniques for each application category.

2 Background

This section provides background and motivation for application-transparent support for memory oversubscription in GPUs. §2.1 provides background on the GPU execution model and unified memory; §2.2 analyzes oversubscription overheads in a real GPU system; §2.3 describes previous methods to avoid oversubscription and motivates the need for a new, application-transparent framework.

2.1 GPU Execution Model

GPUs achieve high throughput via the single instruction multiple thread (SIMT) execution model [66, 93]. In each clock cycle, a GPU core (sometimes referred to as streaming multiprocessor or SM), executes a group of threads, called a *warp* or a *wavefront*. All threads in a warp execute in lockstep. A GPU tolerates long-latency stalls using fine-grained multithreading: each cycle, a different warp is fetched such that no two instructions from the same warp are in the pipeline concurrently. A GPU core stalls when there is no available warp to be executed. Each executing thread can access a different memory location, potentially creating a large number of in-flight, concurrent memory accesses.

Unified Virtual Addressing. Modern GPUs support *unified* virtual address spaces between the host CPU and the

GPU, which allows the CPU to manage data inside GPU physical memory using the same pointers as the ones used by the GPU program [89]. This functionality greatly improves GPU programmability because developers can manage data in both GPU and CPU spaces using the same virtual addresses.

Unified Memory. Even with Unified Virtual Addressing, data in GPU memory and data in CPU memory are still considered to be in separate memory spaces. Developers must programmatically allocate memory on the GPU and copy data from the CPU to the GPU memory before a GPU kernel can access that data. *Unified memory* supports the *abstraction* of a single virtual address space accessible by both CPU programs and GPU kernels [33]. Supporting this abstraction requires automatic demand-driven movement of data between host and GPU memory, and is typically supported by fault-driven transfers at the page or the multi-page (up to 2MB) granularity [102].

2.2 Oversubscription Overheads in GPUs

While unified memory can vastly improve programmability, it is not a panacea. First, the address translation hardware induces performance overheads and can lower GPU throughput. Second, paging of data between the CPU and GPU memories can require frequent high-latency transfers.

While multiple proposals [9, 10, 77, 91] improve the performance of address translation in the GPU (e.g., with parallel page table walks [77], large TLB reach [9] and lower page table walk latency [10]), none of these address the high overhead of demand paging directly. Previous works explore prefetching to hide overheads [102], but they do not consider optimizing performance specifically for cases when GPU memory is oversubscribed.

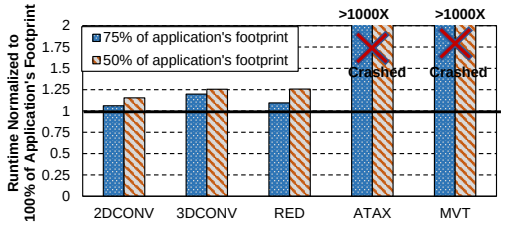


Figure 1. Application runtime sensitivity to GPU memory oversubscription, measured using an NVIDIA GTX 1060.

Figure 1 shows the performance degradation due to memory oversubscription we observe for 5 GPGPU applications from the CUDA SDK [62] and Polybench benchmarks suites [32] when they are run on an NVIDIA GTX 1060 GPU with 2GB available memory [24]. To introduce oversubscription, we manually modify the amount of available memory space assigned to each GPU kernel such that only 50% and 75% of the total memory footprint fit in the GPU's physical memory. We make three observations. First, *all* applications suffer from significant performance loss due to memory oversubscription: the more the memory is oversubscribed, the larger the performance cost. Second, `2DCONV`, `3DCONV` and `RED` suffer from an average 17% performance loss: we find that waiting for eviction of GPU physical pages to create space for newly fetched data is the dominant source of overhead in these workloads. Third, the slowdowns of `ATAX` and `MVT` are larger than 1000x when the GPU memory can hold only 75% of their memory footprint, and both applications crash the entire system when the GPU memory can hold only 50% of their footprint. System crash happens due to thrashing, which moves pages back and forth between CPU and GPU memory repeatedly, dominating the oversubscription overhead in these two workloads.

2.3 An Application-Transparent Framework

Prior Methods to Avoid Oversubscription. Multiple techniques can be used to manage oversubscription. Increasing memory capacity is an efficient way to avoid the oversubscription altogether. On-package 3D-stacked memory (like High-Bandwidth Memory [39, 52] and Hybrid Memory Cube [34, 35]) is widely used in NVIDIA's P100 [67] and V100 GPU [68], AMD Radeon R9 series GPU [8] and Google TPUv2 [31, 43]. However, increasing the memory capacity of on-package 3D stacked memory faces three major challenges. First, the number of stacks is limited by the manufacturing technology. Second, adding more stacks horizontally on the silicon interposer is limited by the wiring complexity of the silicon interposer and the number of pins of chips [57]. Third, as GPGPU application working sets continue to become larger [50, 51], application developers will still need to take the size of GPU memory into account despite the increased capacity. Alternatively, dividing tasks across multiple GPUs or smaller kernels with smaller memory footprint [30, 53] requires non-trivial programming effort to break a complex GPU kernel into multiple GPUs or kernels. Moreover, launching more kernels to a multi-GPU system introduces extra communication complexity among the host CPU and GPU devices.

Naive Designs. To reduce the oversubscription overhead, we perform a design space exploration by employing various mechanisms that aim to reduce the page fault overhead. We evaluate different warp scheduling policies: faulting and non-faulting warps are given different priorities, such that non-faulting warps are prioritized and can still proceed with their data in-memory. However, a warp scheduler that prioritizes the non-faulting warps does *not* reduce the page faults, it only distributes them differently across time. Eventually, *all* threads are stalled waiting for the page faults to complete. We conclude that while warp-level scheduling can be an effective method to hide memory access latency, it is far from enough to hide page fault handling latency, which is orders of magnitude longer than memory latency.

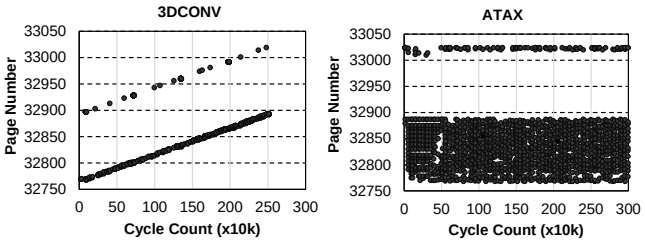
We also experimented with different page replacement policies to enhance locality and minimize thrashing. Conventional wisdom suggests the ideal LRU policy [15] as an

upper bound for achievable performance with page replacement, but this policy is too expensive to implement for large GPU memories. Age-based LRU is easier to implement with a list that can store time when a page is migrated from the CPU memory to the GPU memory [85, 86]. Our measurements show that this age-based LRU policy performs well for applications with streaming access patterns, which we define as regular applications, due to strong sequential locality. However, applications with random access patterns, which we define as irregular applications, do *not* benefit from the age-based LRU policy. In fact, we observed severe thrashing as the working set of these irregular applications becomes larger than the size of GPU physical memory. Hence, no page replacement policy can effectively minimize thrashing.

Objectives. Our goal is threefold. First, our design aims to maximally recover application performance to the non-oversubscribed level, i.e., a system with sufficient memory capacity. Second, the framework should be transparent to the application since we do not want users to manually manage physical memory. Third, our design should be able to address the main performance overhead based on different applications' characteristics.

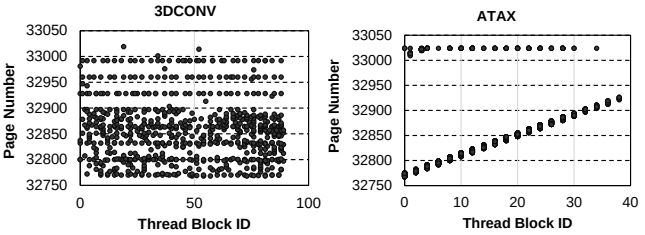
3 Characterizing Memory Accesses for GPGPU Workloads

An effective memory management framework requires understanding of the application memory access behavior, which is dependent on application characteristics. To this end, we first examine the memory access traces of various workloads and extract each workload's most distinctive access patterns. We find that workloads can generally be classified into those with *regular* or *irregular* memory access patterns. Figure 2(a)-(b) show the access pattern of two representative applications (`3DCONV` and `ATAX`). `3DCONV` exhibits a fairly streaming page access pattern across all thread blocks' memory accesses. `ATAX` exhibits rather a random page access pattern across all thread blocks' memory accesses. At any point in time, `3DCONV` accesses only a small number of memory pages. As shown in Figure 3(a), most of its thread blocks access all of the small number of accessed memory pages. In contrast, `ATAX` accesses many memory pages at any point in time. As shown in Figure 3(b), `ATAX`'s thread blocks touch different pages. We observe that many other workloads present a regular memory access pattern similar to `3DCONV`. Such workloads tend to have a relatively small *active page working set*, which is defined as the number of pages that are accessed within a short period of time. In contrast, irregular applications like `ATAX` have much larger active page working sets because each individual thread accesses different pages. This pattern leads to a large number of unique pages to be accessed at a given time. Moreover, we observe that regular applications tend to have more predictable access behavior of their working sets, as indicated by their streaming pattern in Figure 2(a), but irregular applications' access patterns are unpredictable.



(a) Regular Application **(b)** Irregular Application

Figure 2. Example page access patterns of (a) a regular (streaming) application, and (b) an irregular (random access) application.



(a) Regular Application **(b)** Irregular Application

Figure 3. Pages accessed by each thread block in example GPGPU applications: (a) regular, (b) irregular.

We observe that for regular kernels, the memory access pattern is fairly predicable (e.g., streaming). The evicted pages are usually not requested again in the near future, which naturally avoids thrashing. However, it is much harder to predict the access patterns of irregular applications. Once the working set exceeds the memory capacity, any page that is evicted may be requested again, causing thrashing.

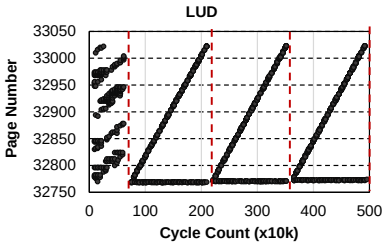


Figure 4. Page access pattern of `LUD`, which is a regular application with data sharing (multiple kernels access the same data; dashed lines represent the end of each kernel).

Data Sharing across Kernels. It is possible that *multiple kernels* access the *same* data in *some* regular applications, as shown in Figure 4. In `LUD`, the memory access pattern of each kernel is streaming with a small number of active pages. However, several kernels in the application share the same data, leading to repeated access by different kernels to the same pages. When the footprint of each kernel is larger than the physical memory size, data migration is required to bring in new pages from the CPU memory, leading to low performance.

We form three conclusions based on our observations. First, the oversubscription overhead for regular applications is mostly eviction overhead. Second, thrashing among different pages dominates the performance overhead in irregular applications. Third, data sharing can incur additional data migration, leading to lower performance. These conclusions guide the design of our proposed framework.

4 The ETC Framework

The key principle of ETC is to use appropriate memory management techniques for different types of applications: 1) *regular applications without data sharing*, 2) *regular applications with data sharing* and 3) *irregular applications*. To this end, the design of ETC comprises four major techniques: Application Classification (AC), Proactive Eviction (PE), Memory-aware Throttling (MT), and memory Capacity Compression (CC).

Upon detecting memory oversubscription, ETC first classifies applications (§4.1). Based on 1) the application type and 2) data sharing behavior across multiple kernels, ETC uses a selection of the PE, MT and CC techniques to reduce the performance overhead of oversubscription. For *regular applications with no data sharing*, ETC employs proactive eviction (§4.2). For *regular applications without data sharing*, ETC employs *both* proactive eviction (§4.2) and capacity compression (§4.4). For *irregular applications*, ETC employs an appropriate amount of SM throttling (§4.3) as well as capacity compression (§4.4).

4.1 Application Classification

Before ETC can select which techniques to employ for which application, it detects 1) the type of application running on each SM and 2) the amount of data sharing between kernels. To detect the type of application running on each SM, ETC uses *memory coalescing* statistics, widely used for profiling applications in SIMT and GPU architectures [20, 38]. When memory requests from the same warp access the same cache line, the memory coalescing unit combines the requests to avoid redundant accesses and thus reduces memory bandwidth consumption. Memory coalescing is prevalent in regular applications [62] due to their high memory access locality. However, it rarely happens in irregular applications due to their poor locality [32, 94]. Based on this observation, ETC employs a counter in each SM's load/store unit to sample the number of coalesced memory accesses. If that number is above a threshold, ETC categorizes the application executing on the SM as a regular application. Otherwise, ETC categorizes the application executing on the SM as an irregular application.

To detect data sharing between kernels, ETC relies on compile-time information. ETC classifies an application as *data sharing* if the compiler detects similar pointer accesses coming from multiple kernels.[1]

[1]ETC utilizes the compiler by marking kernels that contain the same pointer as shared. While this would be prone to aliasing in CPU workloads, GPU workloads are typically written in a way that makes this heuristic accurate the vast majority of the time.

4.2 Proactive Eviction

The key idea of the proactive eviction technique is to preemptively evict pages before the GPU runs out of physical memory. Doing so allows data migration due to page eviction to happen at the same time as data migration due to page faults. Figure 5 provides an example of how our proactive eviction technique works. A failed address translation due to a missing page in physical memory causes a page fault to fetch the page from the host memory as shown in Figure 5(a). If the application exhausts all available physical GPU memory, data access to a new page cannot proceed until another page in GPU memory is evicted back to the host memory first. In current production systems, eviction is triggered only by page faults [33, 67]. Page migration from the CPU to the GPU (host-to-device) *cannot* start until eviction from the GPU to the CPU (device-to-host) is completed, as shown in Figure 5(a). We observe that there is an opportunity to reduce oversubscription eviction overheads by overlapping eviction with page fault handling, as shown in Figure 5(b).

Current GPUs support dual-DMA engines [5, 6, 66, 85], which allows data migration for the page fault and the data migration from the eviction to happen concurrently. Application developers can optimize their programs to overlap the user prefetch and the eviction manually [86]. However, this is still a heavy burden for programmers and directly conflicts with a key goal of on-demand paging: ease of programming. To automatically overlap the eviction with page fault handling, we modify the GPU driver to automatically force pages in GPU memory to be evicted *before* the application runs out of all available physical memory in the GPU. This allows the page fault handling process and the eviction process to occur at the same time.

However, determining the correct timing for proactive eviction is a design challenge for two reasons. First, evicting a page from the GPU too early can cause pages that are still in use to be evicted out of the GPU memory. On the other hand, evicting a page from the GPU memory too late reduces the latency hiding benefit of proactive eviction. Second, the GPU driver must determine how many pages should be evicted at a time. Proactively evicting more pages out of the GPU memory allows the GPU to remove more cold pages and thus make space available for new page faults. However, proactively evicting too many pages can remove hot pages from the GPU memory. We develop a mechanism to achieve a good balance between these tradeoffs.

Avoiding Early Eviction. To determine the correct timing for proactive eviction, we profile various GPGPU applications on a real NVIDIA GTX 1060 and observe how the memory footprint of each application, defined as the number of pages migrated to the GPU, increases over time. Figure 6 shows the number of pages that are migrated from the CPU

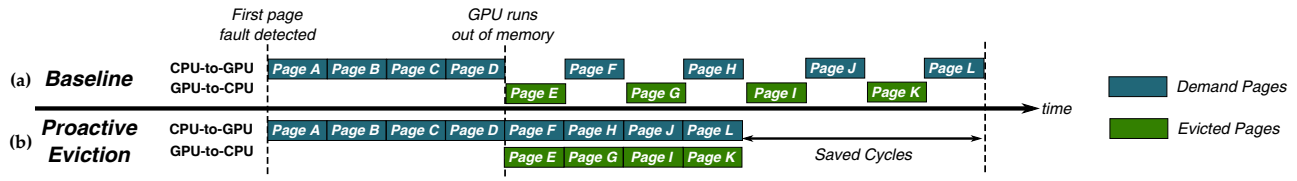


Figure 5. Proactive eviction technique.

memory to the GPU memory for five GPGPU applications. Based on this data, we make four observations. First, the memory footprint increases linearly over time. Second, it is possible that there are multiple phases (observed in the memory footprint of ATAX (blue dot-line shown in Figure 6)), but the trend of the footprint increase rate in each phase is still linear. Third, the nature of GPU's SIMT execution model implies that different warps executing the same instructions can access different data. As all these warps execute in parallel and share the global memory bandwidth, their memory footprint increases until all data is fetched in each phase, which explains the linear increase in memory footprint over time. Fourth, the time interval between page faults is almost constant in each phase. Based on these observations, the GPU can anticipate a series of page faults that are likely to occur within a constant time frame and perform multiple page evictions as soon as the first page fault is detected, in order to create physical memory space for the pages in demand.

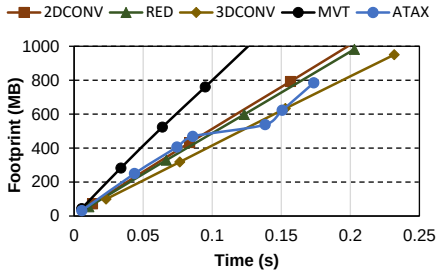


Figure 6. Applications' memory footprint over time.

Avoiding Late Eviction. It is not always the case that the data transfer speed from the CPU to GPU (host-to-device) is similar to that from the GPU to CPU (device-to-host). Via empirical measurements on the NVIDIA GTX 1060 GPU, we found that the data transfer speed from the device to the host is significantly faster than that from the host to the device. Hence, moving the same number of pages from the device (GPU) back to the host (CPU) during eviction can be faster than paging in data from the host (CPU) to the device (GPU). Based on this observation, it is possible for the GPU to avoid late eviction by starting the eviction process at the same time as the occurrence of the page fault.[2]

Irregular applications access a large number of pages within the same time frame (§3). Because of this, proactive eviction becomes ineffective, and we find that the potential downside of thrashing outweighs the potential speed up from the proactive eviction for such applications.

Implementation. To employ proactive eviction, ETC modifies the virtual memory manager inside the GPU runtime to include a new proactive eviction unit (PEU). When a page fault occurs, PEU interrupts the GPU driver so that the GPU driver can move the faulting page into GPU memory. When the GPU driver successfully *allocates* a new page on the GPU memory, PEU starts checking information from the Application Classification logic (§4.1), Then, PEU checks the memory allocation size and compares it with the available memory size to predict if it will be oversubscribed. PEU performs proactive eviction only if 1) the memory allocation size is larger than the available GPU memory size, 2) the GPU memory is oversubscribed and 3) the available memory size is smaller than a threshold (empirically set to 2MB on our evaluation).

4.3 Memory-aware Throttling

As discussed in §2.2, page-level thrashing can significantly degrade the performance of irregular GPGPU applications. As shown in Figures 2(b) and 3(b), a page in an irregular application is accessed only by a few thread blocks. When many thread blocks from irregular applications are executed concurrently on the GPU, the working set size rapidly increases, causing severe thrashing, for which traditional page replacement policies do not provide a solution. To avoid such thrashing, our idea is to limit the number of pages that are accessed simultaneously. To this end, ETC employs Memory-aware Throttling that aims to reduce the working set size of an irregular application by limiting its number of concurrent threads via throttling. GPU throttling can be implemented in two ways: thread block (TB) throttling or SM throttling. TB throttling throttles a fraction of thread blocks within each SM. SM throttling throttles a fraction of SMs in the GPU. We experimented with both and found that TB throttling introduces an overly long adjustment period to reach the level with minimum thrashing. Compared to TB throttling, SM throttling can quickly converge to a state with appropriate working data set size. Hence, ETC utilizes SM throttling to reduce the amount of memory thrashing in GPUs.

Implementation. When an irregular application is detected and the memory is oversubscribed, ETC triggers our epoch-based SM throttling. When throttling is triggered, ETC first throttles half of the SMs by stopping the fetch unit from fetching new instructions (instructions in the pipeline can still be drained). During this initial phase, it is possible that

[2]Note that ETC allows the GPU driver to determine *when* proactive eviction happens based on the observed data transfer latencies between the CPU and the GPU, and vice versa.

ETC throttles too many SMs, leading to underutilization, or throttles too few SMs, leading to thrashing. Hence, ETC adjusts the number of throttled SMs dynamically after the initial phase based on observed memory utilization. As shown in Figure 7, the memory-aware throttling technique divides GPU execution into two epochs: the detection epoch and the execution epoch. During the detection epoch, ETC checks if there is an eviction request or a page fault request to determine the aggressiveness of ETC's SM throttling. The detection epoch ends if a page fault is detected (❶) or the time period for detection expires (❷). Once the detection epoch ends, the memory-aware throttling scheme adjusts the number of active SMs based on the page fault and the page eviction behavior gathered during the detection epoch.

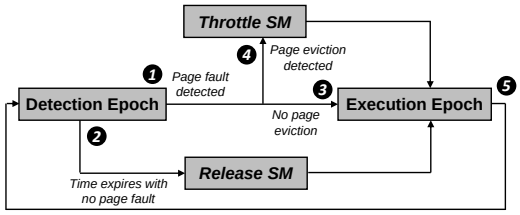


Figure 7. ETC's memory-aware throttling scheme.

If the detection epoch ends because the time period expires (i.e., there is *no page fault*, ❷), it implies that the working set is likely to fit in the GPU memory. The GPU should be able to execute more threads concurrently without page thrashing. In this case, ETC unthrottles an SM. To do this, ETC gradually enables the fetch units in the last throttled SM to increase memory utilization.

If the detection epoch ends because of a *page fault* but *no page eviction occurs* during the detection epoch (❸), it implies that the GPU still has free memory space left.

If the detection epoch ends because of a *page fault* and there is *at least one page eviction (❹), it suggests* that the application's working set size does not fit in the GPU memory and ETC should throttle *more* SMs to reduce the working set size. In this case, as soon as the page fault is resolved, ETC throttles the SM that triggers the page fault, since active warps from this SM are likely to access data that is *not* present in the GPU memory again.

After each adjustment, the GPU begins the execution epoch, which executes all active SMs until the time period for the execution epoch expires and the GPU goes back to the detection epoch again (❺).

With memory-aware throttling, the concurrency of irregular applications can be adjusted so that the working set fits in the available memory. Although the throttling reduces the thread-level parallelism (TLP), we find that it can avoid a lot of memory migrations and recover a significant fraction of the lost performance caused by memory oversubscription. The loss in TLP due to throttling can be recovered by combining ETC's memory-aware throttling with the Capacity Compression technique described in §4.4.

4.4 Capacity Compression

While the proactive eviction technique improves the performance of regular applications and the memory-aware throttling technique improves the performance of irregular applications, there can be cases beyond the two techniques we already discussed, where these techniques alone are insufficient for two reasons. First, ETC's proactive eviction technique can only hide the eviction latency, but it cannot reduce the number of page migrations when pages are shared between multiple kernels.Second, ETC's memory-aware throttling technique is effective at avoiding thrashing in irregular applications, but it comes at the cost of lower thread-level parallelism.

To reduce the impact of memory oversubscription, our goal is to improve the effective capacity of main memory. To this end, we develop a memory compression technique. The key idea behind ETC's capacity compression technique is to selectively apply memory compression when it can lead to performance improvement. Several main memory compression techniques have been proposed [25, 49, 72, 79, 99], and they can be used to increase the effective memory capacity. In this paper, we utilize the Linearly Compressed Pages (LCP) [72] as the framework to compress data in GPU main memory.

LCP is a low-latency main memory compression framework that has been shown to effectively increase memory capacity in a CPU system. We find that LCP can have serious performance impact on a GPU system as it requires additional memory accesses to fetch compression-related metadata that is stored inside the main memory. As shown in Figure 8, the additional accesses to LCP metadata can lead to additional bandwidth demand and can *reduce* the performance of the GPU applications by 13% on average, running on unlimited memory.

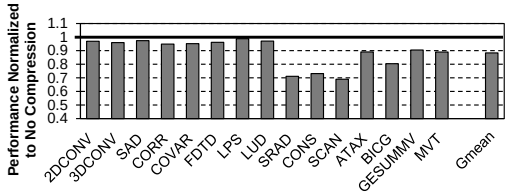


Figure 8. Performance overhead of LCP under unlimited memory.

Hence, it is crucial for ETC to be able to determine when the LCP framework is useful, which happens on two specific classes of applications: *regular applications with data sharing* and *irregular applications*. As thread blocks from both application categories access very large amounts of data, capacity compression allows more data to be stored on the GPU memory. Moreover, the memory-aware throttling technique can be less aggressive when employed together with capacity compression, which leads to a higher TLP than when throttling is used alone.

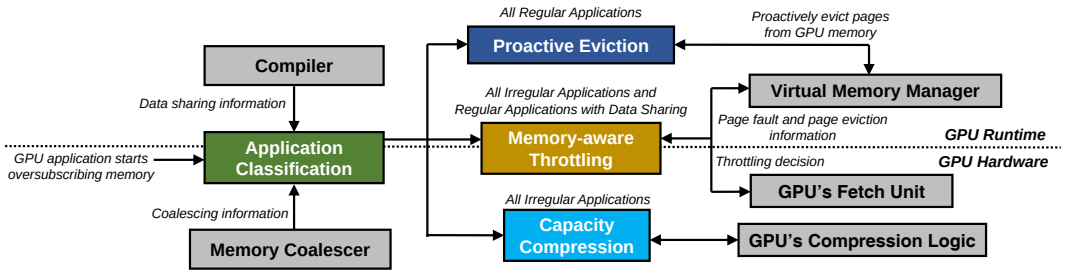


Figure 9. High level overview of ETC showing its four components: Application Classification (**AC**), Proactive Eviction (**PE**), Memory-aware Throttle (**MT**) and Capacity Compression (**CC**).

Implementation. Since modern GPUs already perform memory bandwidth compression within the memory controller and over the PCIe bus [23, 49, 50, 79, 87], both the memory controller and the DMA unit are already equipped with the compression/decompression hardware [23]. To enable LCP, ETC employs an additional 512-entry metadata cache inside the memory controller to accelerate compression metadata lookup and thus reduce the performance overhead of the LCP framework. Once the application classification logic determines that the executing application is 1) a regular application with data sharing or 2) an irregular application, ETC begins the capacity compression process by storing all data written to the GPU memory using the base-delta-immediate compression algorithm [73], which is simple to implement and effective [70–73, 98].

4.5 Design Summary of ETC

Figure 9 shows the design overview of ETC, which consists of Application Classification, Proactive Eviction, Memory-aware Throttling, and memory Capacity Compression.

When the total allocated memory becomes larger than the GPU's physical memory, ETC becomes active and the application classification starts tracking both hardware information on the GPU and gathers the compile-time information. If application classification detects a regular application, ETC enables proactive eviction in the GPU driver's virtual memory manager. ETC also applies the capacity compression to a regular application when data sharing is detected. If the application classification detects an irregular application, ETC performs both memory-aware throttling in order to reduce thrashing and capacity compression to further increase the effective memory capacity.

5 Methodology

We modify the Mosaic simulator [9, 10, 82], which is based on GPGPU-Sim 3.2.2 [11, 40], to evaluate ETC. The configuration of the GPU cores and the memory system are shown in Table 1.

Demand paging and oversubscription. We faithfully model the demand paging of data between the CPU memory and the GPU memory as described in CUDA 8.0 [85, 86]. When a kernel first accesses a page, a TLB miss triggers a page table walk. If the page is not present in GPU memory, the page table walk fails, creating a page fault. The

GPU Core Configurations	
System Overview	30 cores, 64 execution units per core 8 memory partitions
Shader Core Config	1020 MHz, 9-stage pipeline, 64 threads per warp, GTO scheduler [81]
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before accessing L2
Private L1 TLB	64 entries per core, fully associative, LRU
Shared L2 Cache	2MB total, 16-way associative, LRU 2 cache banks 2 interconnect ports per memory partition
Shared L2 TLB	512 entries total, 16-way associative, LRU 2 ports
Page Walk Cache	16-way 8KB
Memory Configurations	
DRAM	GDDR5 1674 MHz, 8 channels 8 banks per rank FR-FCFS scheduler [80, 103], burst length 8
Page Table Walker	64 threads share the page table walker, traversing a 4-level page table
Unified Memory Setup	64KB page size, 2MB maximum eviction size 20μs page fault handler, 16GB/s PCIe bandwidth

Table 1. Configuration of the simulated system.

IOMMU interrupts the CPU to handle page faults. We model an optimistic 20μs page fault latency [102] and employ a state-of-the-art hardware page prefetcher [102] to reduce the page fault overhead. When GPU memory is full, any new page faults must evict old pages using the age-based LRU page replacement policy via the GPU driver [85, 86]. We configure GPU memory capacity to fractions (75% and 50%) of each individual workload's memory footprint on all our experiments except in the ideal *unlimited memory* baseline.

Workloads. We randomly select 15 applications from the CUDA SDK [62], Rodinia [21], Parboil [94] and Polybench [32] benchmarks. We categorized these workloads into three categories: *regular applications with no data sharing* (`2DCONV`, `3DCONV`, `SAD`, `CORR`, `COVAR`, `FDTD` and `LPS`), *regular applications with data sharing* (`LUD`, `SRAD`, `CONS` and `SCAN`), and *irregular applications* (`ATAX`, `BICG`, `GESUMMV` and `MVT`). The footprint of these applications vary from 7.28MB to 70MB with an average of 22.5MB. Impractically long simulation times prevent us from emulating a larger footprint.

Design Parameters. ETC exposes several design parameters. We set the coalescing factors threshold for regular applications to 10 cache lines for our application classification technique. We set 2MB of remaining GPU memory space as the threshold to trigger the proactive eviction techniques. We set both throttling degree and releasing degree to 1 SM at a time as we empirically find that this value yields the highest performance.

6 Evaluation

We evaluate ETC by comparing it against 1) a state-of-the-art realistic baseline (BL) that uses page prefetching [102], and 2) an ideal baseline with unlimited amount of DRAM.

6.1 Performance

Figure 10 shows the performance of ETC across different workload categories normalized to the baseline where the GPU has unlimited memory. We make three conclusions. First, ETC is effective at reducing the performance overhead of oversubscription and performs similarly to the unlimited memory baseline for *regular applications with no data sharing* because page eviction latency, which is fully hidden by our proactive eviction technique, is the major performance overhead for these applications. Second, we find that page migration due to the synchronization between different kernels cannot be avoided for *regular applications with data sharing*. However, ETC still improves performance by an average of 60.4% for such applications compared to the state-of-the-art design. Third, ETC improves the performance of irregular applications by 2.7× compared to the state-of-the-art BL. We conclude that our ETC framework is effective at reducing the performance impact of oversubscription.

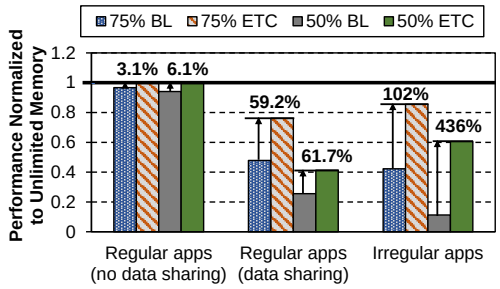


Figure 10. Performance normalized to a GPU with unlimited memory.

6.2 Analysis of Techniques and Workloads

We provide an in-depth analysis of how each technique of ETC affects the performance of each workload type.

Regular Applications with no Data Sharing. Figure 11 shows the impact of proactive eviction (PE) and capacity compression (CC) on regular applications with no data sharing. We make three observations. First, when proactive eviction becomes active, the eviction latency can almost always be completely overlapped with the page fault latency. Among all the regular applications with no data sharing that we evaluated, eviction latency cannot be completely overlapped in only one application (LPS) where we find ETC evicts pages too aggressively.

Second, while not shown in Figure 11 due to space constraints, *regular applications with no data sharing* do *not* benefit from SM throttling because SM throttling does not hide eviction latency. In fact, SM throttling decreases TLP, and thus the latency hiding capability.

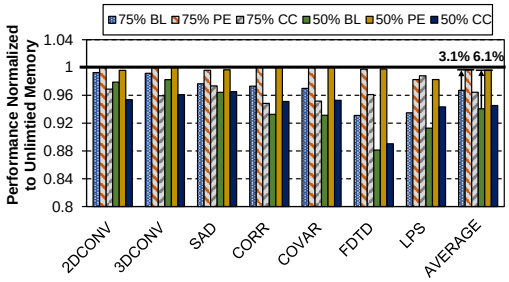


Figure 11. Performance of regular applications with no data sharing.

Third, *regular applications with no data sharing* perform worse than the state-of-the-art baseline when capacity compression (CC) is applied, due to the additional accesses to compression metadata, as discussed in §4.4.

Regular Applications with Data Sharing. Figure 12 shows the performance of *regular applications with data sharing* across kernels when proactive eviction (PE), capacity compression (CC) and both (PE+CC) are applied. We make four observations. First, compared to the unlimited memory baseline, the state-of-the-art BL suffers 52.2% (74.1%) performance loss when GPU memory can fit only 75% (50%) of the applications' memory footprint. Second, the average performance of the proactive eviction technique alone (PE) is only 9.3% better than that of the state-of-the-art baseline (BL) because additional data migration, due to data sharing, dominates the oversubscription overhead for this type of applications. Third, the capacity compression mechanism alone (CC) yields 52.8% average performance improvement over the state-of-the-art baseline, due to the increased effective memory capacity. Fourth, combined proactive eviction and capacity compression (PE+CC) results in 60.4% average performance improvement.

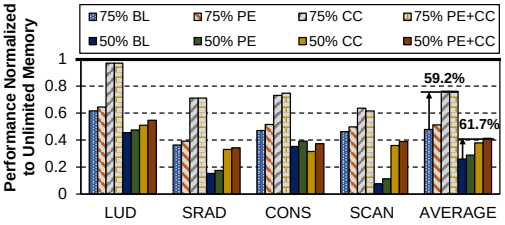


Figure 12. Performance of regular applications with proactive eviction and capacity compression.

We conclude that ETC improves the performance of regular GPU applications regardless of whether or not the data is shared across kernels.

Irregular Applications. Figures 13 and 14 show the performance and total eviction counts of each individual component of ETC on *irregular applications*.

To evaluate ETC's throttling scheme (MT), we compare it against a naive throttling scheme that statically throttles half of the SMs at the beginning of execution (denoted as NT in Figures 13 and 14). We make three observations. First, the

naive scheme (NT) outperforms the state-of-the-art baseline design (BL) by 57.7% when 75% of the footprint fits in memory. Second, when GPU memory is more limited at 50% of the footprint, the naive scheme (NT) is ineffective and degrades performance by 10.5% compared to the BL. In contrast, our memory-aware throttling scheme (MT), which dynamically adjusts how many SMs to throttle, provides 436% performance improvement over BL. Third, our adaptive throttling scheme (MT) performs worse than naive throttling (NT) on two workloads (BICG and GESUMMV) in the scenario when 75% of the memory footprint fits in the memory capacity, because of the adjustment latency to reach the appropriate number of active SMs to be throttled.

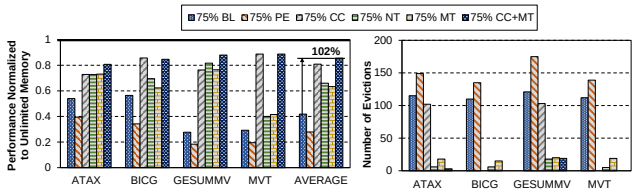


Figure 13. Performance of irregular applications (75% of applications' memory footprint fits in memory).

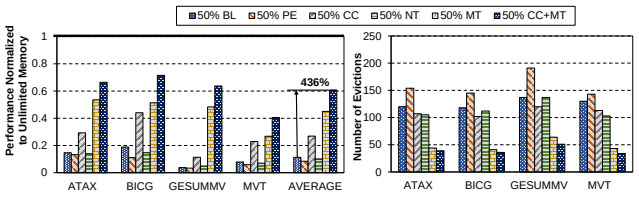


Figure 14. Performance of irregular applications (50% of applications' memory footprint fits in memory).

Figure 15 shows the page fault rate of an irregular application (ATAX) over 10 million cycles. When the memory is oversubscribed and only 75% of the footprint fits in memory, thrashing ensues and frequent page faults occur. In contrast, when the memory-aware throttling mechanism (MT) is active, page faults are infrequent, indicating that MT is effective at decreasing the working set. Moreover, fewer evictions occur with MT, as shown in Figures 13 and 14.

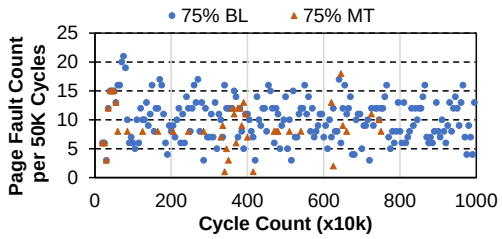


Figure 15. Page fault rate of ATAX.

The performance benefit of capacity compression in irregular applications is determined by both the compression ratio and the size of GPU's physical memory relative to the application's memory footprint. When the entire memory footprint of an application fits in GPU memory after compression, page faults no longer occur. The significant reduction in the number of page evictions for BICG and MVT, shown in Figure 13, suggests that the compression ratios of these two applications are high enough to fit almost all of their working sets in main memory, leading to 51.8% and 203.6% performance improvement over BL, respectively. The performance of BICG and MVT recovers to 85.7% and 88.7% of their ideal unlimited memory performance, respectively. Figure 14 shows a scenario where the GPU's physical memory is much more limited (50% of applications' memory footprint). All applications suffer from thrashing even after capacity compression (CC) is employed alone. MT reduces thrashing and, together with CC, improves performance by 436% compared to the state-of-the-art BL. We conclude that while capacity compression can improve the performance of oversubscribed irregular GPGPU applications, page faults can still remain and hinder performance. Thus, the combination of capacity compression and memory-aware throttling is especially desirable at high levels of memory oversubscription.

We observe that the proactive eviction scheme (PE) causes an average performance loss of 29.7% over reactive eviction (BL) as pages are prematurely evicted from the GPU's physical memory. Thus, PE is not a good technique for irregular applications, as we discussed earlier.

In summary, irregular applications significantly benefit from memory-aware throttling (MT) and capacity compression (CC). Thus, the ETC framework uses both schemes to achieve good performance. As shown in Figures 13 and 14, ETC (CC+MT) increases performance by 270%, on average, for irregular applications. Although throttling decreases TLP, it is able to effectively reduce oversubscription overheads and thrashing.

6.3 Classification Accuracy

As discussed in §6.2, ETC relies on correct classification of the type of application executing on the GPU to select the best scheme (See §4.1). Figure 16 compares the average sampled coalescing factor over 50k cycles and the actual coalescing factor of each application. We can observe a large gap between coalescing factors of regular applications and irregular applications. We find that any threshold value between 5 and 10 enables the accuracy of ETC's application classification to be 100%. We set the coalescing factor threshold to 10.

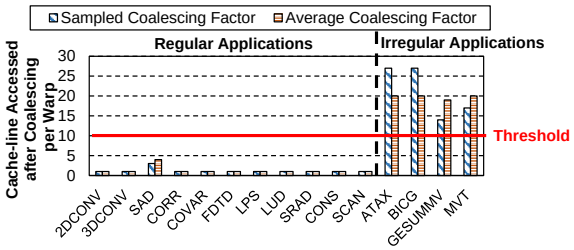


Figure 16. Measured coalescing factors (at the cache-line level) for different applications.

Figure 17 shows the average number of pages accessed by instructions from each warp. A warp from irregular applications typically accesses multiple pages at once while almost all warps from regular applications access only one page at a time. Hence, using the coalescing factor at the page level is as effective as using the coalescing factor at the cache-line level to distinguish between different types of applications.

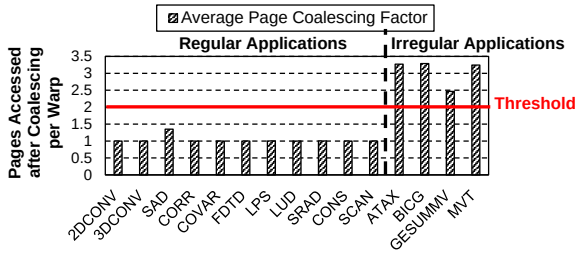
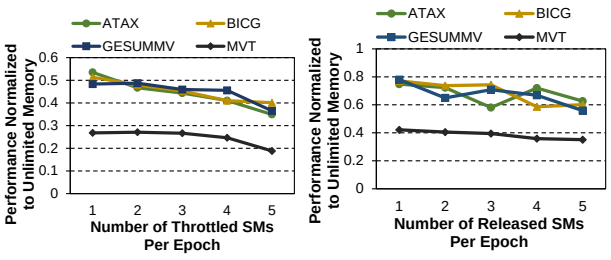


Figure 17. Measured coalescing factors (at the page level) for different applications

6.4 Sensitivity Analysis

In this section, we measure the sensitivity of ETC's performance to the aggressiveness of the memory-aware throttling scheme, the page fault handling latency and the size of DRAM on the GPU.

SM Throttling Aggressiveness. The number of SMs that are throttled and released per epoch can affect application performance. Figure 18 shows normalized performance when we vary the number of SMs that are throttled (fewer active GPU cores) or released (more active GPU cores) per epoch. Based on Figure 18, we make two observations. First, our memory-aware throttling scheme achieves the highest performance when both the throttle degree and the release degree are 1, suggesting that fine-grained adjustment works well. Second, we observe that performance is more sensitive to SM throttling aggressiveness than release aggressiveness because page-faults have a larger negative effect on performance than the reduction of TLP does.



(a) Throttle Degree **(b)** Release Degree

Figure 18. Performance vs. SM throttling aggressiveness.

Fault Latency. Figure 19 shows the performance of GPGPU applications when page fault latency is varied from 20μs to 50μs normalized to when the page fault latency is 20μs. We observe that average performance drops 31.2% when the fault latency increases from 20μs to 50μs. This data shows that hiding the eviction latency becomes important for recovering the performance loss due to oversubscription as page faults become a more dominant source of the performance bottleneck.

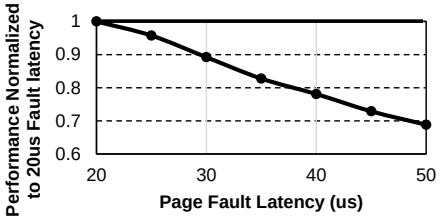


Figure 19. Performance of ETC vs. page fault latency.

Compression Ratio. The compression ratio of each application's memory footprint affects the number of pages that fits in the GPU main memory. Figure 20 shows the average performance of all workloads using various synthetic compression ratios[3], normalized to the performance of each application with no compression when the GPU physical memory capacity is set to 50% of each application's working set size. The data shows that GPU performance increases almost linearly as more compressed data fits in the GPU memory, and performance significantly improves when *all* application data fits in the GPU memory (at the compression ratio of 2).

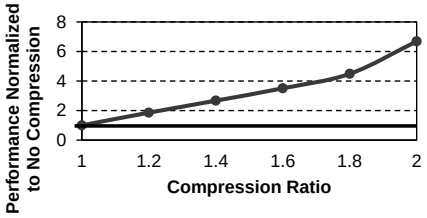


Figure 20. Sensitivity of performance to compression ratio.

6.5 Hardware Overhead

We analyze the hardware overhead to support each component of ETC. Proactive eviction does not require any hardware overhead and is implemented in the GPU driver. We modify the driver to detect the available memory size and trigger eviction proactively. To implement memory-aware throttling, the IOMMU must be extended to implement our throttling adjustment scheme. Two 32-bit counters are added to track epochs. Control logic is added to disable fetch units. To implement capacity compression, we add similar hardware extensions to those required by the LCP framework for CPUs, which consists of a 512-entry metadata cache. No additional hardware is needed for compression as it is already

[3]We include the compression and decompression overheads in performance simulations.

available on current GPUs and the (de)compression units already exist in the memory controller as well [49, 79, 98]. We extend the page table entry, using 9 bits to include page compression information. Finally, the application classifier requires: (1) a 32-bit coalescing factor counter in each load/store unit; (2) signals to fetch units, compression units and the IOMMU.

Overall, hardware overheads for our design are modest. In addition to the logic overhead, the storage overhead is the 32KB metadata cache and 482 32-bit counters (16 counters in each of 30 SMs and 2 counters in the IOMMU), which is less than 2KB of storage cost.

7 Related Work

To our knowledge, this paper is the first to propose an application-transparent hardware/software cooperative solution that uses the most effective combination of techniques for each application category. We survey previous techniques that aim to (1) provide unified virtual memory support on GPUs, (2) reduce the overhead of memory oversubscription, (3) achieve good thread-level parallelism, and (4) increase the effective memory capacity.

GPU Virtual Memory. Address translation overheads are well-studied for CPUs [3, 4, 12, 14, 16–18, 27–29, 44, 45, 58, 60, 69, 74, 75, 88, 92, 95]. For GPUs, Pichai et al. [76] and Power et al. [77] explore IOMMU designs to improve the throughput of address translation based on GPU memory access patterns. Cong et al. [22] propose TLB support for a unified virtual address space between the host CPU and customized accelerators. MASK [10] is a TLB-aware GPU memory hierarchy design that prioritizes memory metadata accesses (e.g., page walks) over data accesses, to accelerate address translation. Mosaic [9] provides application-transparent multiple page size support in GPUs to increase TLB reach. Shin et al. [91] propose a SIMT-aware mechanism to improve address translation performance in irregular GPU workloads.

On-demand Paging. Traditionally, GPGPU memory footprint has been limited by physical memory capacity [63–65], with kernel launch delayed until all required CPU-GPU data transfer completes. Modern GPUs automate GPU memory management [8, 67]: pages are moved to/from GPU memory on-demand, and kernel execution overlaps data transfer, reducing programmer effort and enabling workloads with large memory footprint. Zheng et al. [102] explore migration overheads and propose programmer-directed memory management to hide overheads. Their technique is orthogonal to our work and we apply it as the baseline technique in all our configurations, including ETC.

GPU Memory Oversubscription. GPUswap [48] enables GPU memory oversubscription by relocating GPU application data to CPU memory, keeping data accessible from the GPU. GPUswap provides basic oversubscription support but does *not* reduce oversubscription overheads. The VAST runtime [53] partitions data-parallel workloads based on available GPU physical memory but requires programmer-driven code transformations. The BW-AWARE [2] page placement policy uses heterogeneous memory system characteristics and annotations to guide data placement, focusing on a globally-addressable heterogeneous memory system. Our work reduces oversubscription overheads transparently.

GPU TLP Management. Previous designs [26, 41, 42, 46, 47, 54, 55, 61, 81, 90, 100, 101] control the parallelism of GPU cores to achieve high TLP and high performance. Rogers et al. [81] propose an adaptive HW mechanism to limit TLP to avoid L1 thrashing. Kayiran et al. [47] propose a dynamic CTA scheduling mechanism to modulate the core-level TLP, which reduces memory resource contention. Mascar [90] detects memory saturation and prioritizes memory requests among warps. Wang et al. [100] propose pattern-based TLP management that modulates TLP of concurrent applications. Our work reduces the effective memory working set under oversubscription, which none of these works does.

Memory Compression in GPUs. Several works study memory and cache compression in GPUs [49, 70, 71, 79, 87, 99]. These works show benefits due to on-chip and off-chip memory bandwidth savings. We demonstrate that capacity compression in GPUs is beneficial in certain cases, and develop a mechanism that decides when to use compression.

8 Conclusion

We introduce ETC, an application-transparent framework for reducing memory oversubscription overheads in GPUs. Regular and irregular applications exhibit different types of behavior when memory is oversubscribed. Regular applications are most affected by page eviction latency, while irregular ones are prone to memory thrashing. ETC classifies applications as regular and irregular, and uses 1) *proactive eviction* to hide the page eviction latency, 2) *memory-aware throttling* to ameliorate thrashing, and 3) *capacity compression* to increase the effective memory capacity. For regular applications with no data sharing, ETC eliminates the overhead of memory oversubscription and performs similar to the *ideal* unlimited memory baseline. For regular applications with data sharing and irregular applications, ETC improves the performance by 60.4% and 270% compared with the state-of-the-art baseline. We conclude that ETC is an effective low-cost framework to minimize memory oversubscription overheads in modern GPU systems.

Acknowledgments

We thank the anonymous reviewers from ASPLOS 2019. We acknowledge the support of our industrial partners, especially Google, Intel, Microsoft, and VMware. This research is partially supported by the NSF (grants 1409723, 1422331, 1617071, 1618563, 1657336, 1718080, 1725657, and 1750667), the National Natural Science Foundation of China (61832018) and the Semiconductor Research Corporation. This work was carried out while Chen Li visited the University of Pittsburgh on a CSC scholarship.

References

[1] Advanced Micro Devices, Inc. 2013. What is Heterogeneous System Architecture (HSA)? http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/

[2] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *ASPLOS*.

[3] J. Ahn, S. Jin, and J. Huh. 2012. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *ISCA*.

[4] J. Ahn, S. Jin, and J. Huh. 2015. Fast Two-Level Address Translation for Virtualized Systems. *IEEE TC* (2015).

[5] AMD. 2011. AMD Accelerated Processing Units. http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx.

[6] AMD. 2012. AMD Graphics Cores Next (GCN) Architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[7] AMD 2016. *AMD I/O Virtualization Technology (IOMMU) Specification*. AMD. http://support.amd.com/TechDocs/48882_IOMMU.pdf

[8] AMD. 2017. Radeon's Next-generation Vega Architecture. https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf.

[9] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. Rossbach, and O. Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *MICRO*.

[10] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. Rossbach, and O. Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *ASPLOS*.

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

[12] T. W. Barr, A. L. Cox, and S. Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *ISCA*.

[13] T. W. Barr, A. L. Cox, and S. Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *ISCA*.

[14] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *ISCA*.

[15] L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal* (1966).

[16] A. Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *MICRO*.

[17] A. Bhattacharjee, D. Lustig, and M. Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *HPCA*.

[18] A. Bhattacharjee and M. Martonosi. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *PACT*.

[19] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. 1989. Translation Lookaside Buffer Consistency: A Software Approach. In *ASPLOS*.

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

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*.

[22] J. Cong, Z. Fang, Y. Hao, and G. Reinmana. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *HPCA*.

[23] N. Corp. 2015. NVIDIA Tegra X1: NVIDIA's New Mobile Superchip. http://www.nvidia.com/object/tegra-x1-processor.html.

[24] N. Corp. 2016. NVIDIA GTX 1060. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1060/.

[25] M. Ekman and P. Stenstrom. [n. d.]. A Robust Main-Memory Compression Scheme.

[26] W. Fung, I. Sham, G. Yuan, and T. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*.

[27] J. Gandhi, A. Basu, M. Hill, and M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *MICRO*.

[28] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *MICRO*.

[29] J. Gandhi, M. D. Hill, and M. M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *ISCA*.

[30] N. Gawande, J. Daily, C. Siegel, N. Tallent, and A. Vishnu. 2018. Scaling Deep Learning Workloads: NVIDIA DGX-1/Pascal and Intel Knights Landing. *Future Generation Computer Systems* (2018).

[31] Google. 2017. Cloud TPUs: ML Accelerators for TensorFlow. *https://cloud.google.com/tpu/* (2017).

[32] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a High-level Language Targeted to GPU Codes. In *InPar*.

[33] M. Harris. 2013. Unified Memory in CUDA 6. https://devblogs.nvidia.com/unified-memory-in-cuda-6/

[34] Hybrid Memoty Cube Consortium. 2013. HMC Specification 1.1.

[35] Hybrid Memoty Cube Consortium. 2014. HMC Specification 2.0.

[36] IBM. 2017. Realizing the value of Large Model Support (LMS) with PowerAI IBM Caffe. http://developer.ibm.com/linuxonpower/2017/09/22/realizing-value-large-model-support-lms-powerai-ibm-caffe/.

[37] Intel 2014. *Intel Virtualization Technology for Directed I/O*. Intel. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf

[38] B. Jang, D. Schaa, F. Mistry, and D. Kaeli. 2010. Exploiting Memory Access Patterns to Improve Memory Performance in Data-parallel Architectures. *IEEE TPDS* (2010).

[39] JEDEC. 2018. High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235a.

[40] A. Jog, O. Kayıran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. Keckler, M. Kandemir, and C. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *MEMSYS*.

[41] A. Jog, O. Kayıran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*.

[42] A. Jog, O. Kayıran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*.

[43] N. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, and et.al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*.

[44] G. B. Kandiraju and A. Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-Driven Study. In *ISCA*.

[45] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *ISCA*.

[46] O. Kayıran, N. Chidambaram, A. Jog, R. Ausavarungnirun, M. Kandemir, G. Loh, O. Mutlu, and C. Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *MICRO*.

[47] O. Kayıran, A. Jog, M. Kandemir, and C. Das. 2013. Neither More nor Less: optimizing Thread-level Parallelism for GPGPUs. In *PACT*.

[48] J. Kehne, J. Metter, and F. Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping. In *VEE*.

[49] J. Kim, M. Sullivan, E. Choukse, and M. Erez. 2016. Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures. In *ISCA*.

[50] Y. Kwon and M. Rhu. 2018. A Case for Memory-Centric HPC System Architecture for Training Deep Neural Networks. *IEEE CAL* (2018).

[51] Y. Kwon and M. Rhu. 2018. Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning. In *MICRO*.

[52] D. Lee, G. Pekhimenko, S. M. Khan, S. Ghose, and O. Mutlu. 2016. Simultaneous Multi Layer Access: A High Bandwidth and Low Cost

3D-Stacked Memory Interface. In *ACM TACO*.
[53] J. Lee, M. Samadi, and S. Mahlke. 2014. VAST: The Illusion of a Large Memory Space for GPUs. In *PACT*.
[54] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. 2015. Locality-Driven Dynamic GPU Cache Bypassing. In *ICS*.
[55] X. Li and Y. Liang. 2016. Efficient Kernel Management on GPUs. In *DATE*.
[56] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008).
[57] G. Loh, N. Jerger, A. Kannan, and Y. Eckert. 2015. Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems. In *MEMSYS*.
[58] D. Lustig, A. Bhattacharjee, and M. Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. In *ACM TACO*.
[59] C. Meng, M. Sun, J. Yang, M. Qiu, and Y. Gu. 2017. Training deeper models by GPU memory optimization on TensorFlow. In *Proc. of ML Systems Workshop in NIPS*.
[60] T. Merrifield and H. Taheri. 2016. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *VEE*.
[61] V. Narasiman et al. 2011. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. *MICRO*.
[62] NVIDIA Corp. 2011. CUDA C/C++ SDK Code Samples. http://developer.nvidia.com/cuda-cc-sdk-code-samples.
[63] NVIDIA Corp. 2011. CUDA Toolkit 4.0. https://developer.nvidia.com/cuda-toolkit-40.
[64] NVIDIA Corp. 2012. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.
[65] NVIDIA Corp. 2014. NVIDIA GeForce GTX 750 Ti. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.
[66] NVIDIA Corp. 2015. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[67] NVIDIA Corp. 2016. NVIDIA Tesla P100 P100 GPU Architecture. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.
[68] NVIDIA Corp. 2016. NVIDIA Tesla V100 GPU Architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.
[69] M. M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. 2015. Prediction-Based Superpage-Friendly TLB Designs. In *HPCA*.
[70] G. Pekhimenko, E. Bolotin, M. OâĂŹConnor, O. Mutlu, T. C. Mowry, and S. W. Keckler. 2015. Toggle-Aware Compression for GPUs. *IEEE CAL*.
[71] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. Mowry, and S. Keckler. 2016. A Case for Toggle-aware Compression for GPU Systems. In *HPCA*.
[72] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. Kozuch, P. Gibbons, and T. Mowry. 2013. Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency. In *MICRO*.
[73] G. Pekhimenko, V. Seshadri, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry. 2012. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *PACT*.
[74] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *HPCA*.
[75] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *MICRO*.
[76] B. Pichai, L. Hsu, and A. Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS*.
[77] J. Power, M. Hill, and D. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *HPCA*.
[78] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *MICRO*.
[79] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *HPCA*.
[80] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. 2000. Memory Access Scheduling. In *ISCA*.
[81] T. Rogers, M. O'Connor, and T. Aamodt. 2012. Cache-conscious Wavefront Scheduling. In *MICRO*.
[82] SAFARI Research Group. 2017. Mosaic – GitHub Repository. https://github.com/CMU-SAFARI/Mosaic/.
[83] N. Sakharnykh. 2016. Beyond GPU Memory Limits with Unified Memory on Pascal. https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/
[84] N. Sakharnykh. 2017. Maximizing Unified Memory Performance in CUDA. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/
[85] N. Sakharnykh. 2017. Unified Memory on Pascal and Volta. In *NVIDIA GTC*.
[86] N. Sakharnykh. 2018. Everything You Need to Know About Unified Memory. In *NVIDIA GTC*.
[87] V. Sathish, M. Schulte, and N. Kim. 2012. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *PACT*.
[88] A. Saulsbury, F. Dahlgren, and P. Stenström. 2000. Recency-Based TLB Preloading. In *ISCA*.
[89] T. Schroeder. 2011. Peer-to-Peer & Unified Virtual Addressing. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf
[90] A. Sethia, D. Jamshidi, and S. Mahlke. 2015. Mascar: Speeding Up GPU Warps by Reducing Memory Pitstops. In *HPCA*.
[91] S. Shin, G. Cox, M. Oskin, G. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *ISCA*.
[92] S. Srikantaiah and M. Kandemir. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *MICRO*.
[93] J. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* (2010).
[94] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. Univ. of Illinois at Urbana-Champaign.
[95] M. Talluri and M. D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *ASPLOS*.
[96] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee. 2016. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*.
[97] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. Gibbons, and O. Mutlu. 2016. Zorua: A Holistic Approach to Resource Virtualization in GPUs. In *MICRO*.
[98] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. Mowry, and O. Mutlu. 2015. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *ISCA*.
[99] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. 2015. A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 41–53.
[100] H. Wang, F. Luo, M. Ibrahim, O. Kayıran, and A. Jog. 2018. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *HPCA*.

[101] P. Xiang, Y. Yang, and H. Zhou. 2014. Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation. In *HPCA*.

[102] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *HPCA*.

[103] W. Zuravleff and T. Robinson. 1997. Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order.