MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency

Rachata Ausavarungnirun[1] Vance Miller[2] Joshua Landgraf[2] Saugata Ghose[1]
Jayneel Gandhi[3] Adwait Jog[4] Christopher J. Rossbach[2,3] Onur Mutlu[5,1]

[1]Carnegie Mellon University [2]University of Texas at Austin [3]VMware Research
[4]College of William and Mary [5]ETH Zürich

Abstract

Graphics Processing Units (GPUs) exploit large amounts of thread-level parallelism to provide high instruction throughput and to efficiently hide long-latency stalls. The resulting high throughput, along with continued programmability improvements, have made GPUs an essential computational resource in many domains. Applications from different domains can have vastly different compute and memory demands on the GPU. In a large-scale computing environment, to efficiently accommodate such wide-ranging demands without leaving GPU resources underutilized, multiple applications can *share* a single GPU, akin to how multiple applications execute concurrently on a CPU. Multi-application concurrency requires several support mechanisms in both hardware and software. One such key mechanism is *virtual memory*, which manages and protects the address space of each application. However, modern GPUs lack the extensive support for multi-application concurrency available in CPUs, and as a result suffer from high performance overheads when shared by multiple applications, as we demonstrate.

We perform a detailed analysis of which multi-application concurrency support limitations hurt GPU performance the most. We find that the poor performance is largely a result of the virtual memory mechanisms employed in modern GPUs. In particular, poor address translation performance is a key obstacle to efficient GPU sharing. State-of-the-art address translation mechanisms, which were designed for single-application execution, experience significant *inter-application interference* when multiple applications spatially share the GPU. This contention leads to frequent misses in the shared translation lookaside buffer (TLB), where a single miss can induce long-latency stalls for hundreds of threads. As a result, the GPU often cannot schedule enough threads to successfully hide the stalls, which diminishes system throughput and becomes a first-order performance concern.

Based on our analysis, we propose *MASK*, a new GPU framework that provides low-overhead virtual memory support for the concurrent execution of multiple applications. *MASK* consists of three novel address-translation-aware cache and memory management mechanisms that work together to largely reduce the overhead of address translation: (1) a token-based technique to reduce TLB contention, (2) a bypassing mechanism to improve the effectiveness of cached address translations, and (3) an application-aware memory scheduling scheme to reduce the interference between address translation and data requests. Our evaluations show that *MASK* restores much of the throughput lost to TLB contention. Relative to a state-of-the-art GPU TLB, *MASK* improves system throughput by 57.8%, improves IPC throughput by 43.4%, and reduces application-level unfairness by 22.4%. *MASK*'s system throughput is within 23.2% of an ideal GPU system with no address translation overhead.

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

Keywords graphics processing units, GPGPU applications, address translation, virtual memory management, memory protection, memory interference, memory systems, performance

ACM Reference format:
R. Ausavarungnirun et al. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, USA, March 24–28, 2018 (ASPLOS'18),* 16 pages.

1 Introduction

Graphics Processing Units (GPUs) provide high throughput by exploiting a high degree of *thread-level parallelism*. A GPU executes hundreds of threads concurrently, where the threads are grouped into multiple *warps*. The GPU executes each warp in *lockstep* (i.e., each thread in the warp executes the *same* instruction concurrently). When one or more threads of a warp stall, the GPU hides the latency of this stall by scheduling and executing another warp. This high throughput provided by a GPU creates an opportunity to accelerate applications from a wide range of domains (e.g., [1, 11, 28, 31, 37, 45, 65, 73, 80, 82, 85, 91, 121]).

GPU compute density continues to increase to support demanding applications. For example, emerging GPU architectures are expected to provide as many as 128 *streaming multiprocessors* (i.e., GPU cores) per chip in the near future [13, 135]. While the increased compute density can help many individual general-purpose GPU (GPGPU) applications, it exacerbates a growing need to *share* the GPU cores *across multiple applications* in order to fully utilize the large amount of GPU resources. This is especially true in large-scale computing environments, such as cloud servers, where diverse demands for compute and memory exist across different applications. To enable efficient GPU utilization in the presence of application heterogeneity, these large-scale environments rely on the ability to *virtualize* the GPU compute resources and execute multiple applications *concurrently* on a single GPU [7, 9, 50, 51].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03...$15.00
https://doi.org/10.1145/3173162.3173169

The adoption of GPUs in large-scale computing environments is hindered by the primitive virtualization support in contemporary GPUs [3, 4, 5, 12, 27, 34, 49, 53, 92, 93, 95, 96, 98, 101, 107, 139, 142]. While hardware virtualization support has improved for integrated GPUs [3, 12, 27, 34, 49, 53, 92, 93, 107, 142], where the GPU cores and CPU cores are on the same chip and share the same off-chip memory, virtualization support for discrete GPUs [4, 5, 12, 95, 96, 98, 101, 107, 139], where the GPU is on a different chip than the CPU and has its own memory, is insufficient. Despite poor existing support for virtualization, discrete GPUs are likely to be more attractive than integrated GPUs for large-scale computing environments, as they provide the highest-available compute density and remain the platform of choice in many domains [1, 11, 28, 31, 45].

Two alternatives for virtualizating discrete GPUs are *time* multiplexing and *spatial* multiplexing. Modern GPU architectures support time multiplexing using application preemption [44, 79, 96, 101, 127, 141], but this support currently does *not* scale well because each additional application increases contention for the limited GPU resources (Section 2.1). Spatial multiplexing allows us to share a GPU among concurrently-executing applications much as we currently share multi-core CPUs, by providing support for *multi-address-space concurrency* (i.e., the concurrent execution of application kernels from different processes or guest VMs). By efficiently and dynamically managing application kernels that execute concurrently on the GPU, spatial multiplexing *avoids* the scaling issues of time multiplexing. To support spatial multiplexing, GPUs must provide architectural support for *both* memory virtualization and memory protection.

We find that existing techniques for spatial multiplexing in modern GPUs (e.g., [96, 101, 102, 103]) have two major shortcomings. They either (1) require significant programmer intervention to adapt existing programs for spatial multiplexing; or (2) sacrifice memory protection, which is a key requirement for virtualized systems. To overcome these shortcomings, GPUs must utilize *memory virtualization* [54], which enables multiple applications to run concurrently while providing memory protection. While memory virtualization support in modern GPUs is also primitive, in large part due to the poor performance of address translation, several recent efforts have worked to improve address translation within GPUs [35, 105, 106, 134, 151]. These efforts introduce *translation lookaside buffer* (TLB) designs that improve performance significantly when a single application executes on a GPU. Unfortunately, as we show in Section 3, even these improved address translation mechanisms suffer from high performance overheads during spatial multiplexing, as the limited capacities of the TLBs become a source of significant contention within the GPU.

In this paper, we perform a thorough experimental analysis of concurrent multi-application execution when state-of-the-art address translation techniques are employed in a discrete GPU (Section 4). We make three *key observations* from our analysis. First, a single TLB miss frequently stalls *multiple* warps at once, and incurs a very high latency, as each miss must walk through multiple levels of a page table to find the desired address translation. Second, due to high contention for shared address translation structures among the multiple applications, the TLB miss rate increases significantly. As a result, the GPU often does *not* have enough warps that are ready to execute, leaving GPU cores idle and defeating the GPU's latency hiding properties. Third, contention between applications induces significant thrashing on the shared L2 TLB and significant interference between TLB misses and data requests throughout the entire GPU memory system. With only a few simultaneous TLB miss requests, it becomes difficult for the GPU to find a warp that can be scheduled for execution, which defeats the GPU's basic fine-grained multithreading techniques [119, 120, 128, 129] that are essential for hiding the latency of stalls.

Based on our extensive experimental analysis, we conclude that *address translation is a first-order performance concern* in GPUs when multiple applications are executed concurrently. **Our goal** in this work is to develop new techniques that can alleviate the severe address translation bottleneck in state-of-the-art GPUs.

To this end, we propose **M**ulti-**A**ddress **S**pace Concurrent **K**ernels (*MASK*), a new GPU framework that minimizes inter-application interference and address translation overheads during concurrent application execution. The overarching idea of *MASK* is to make the entire memory hierarchy *aware of TLB requests*. *MASK* takes advantage of locality across GPU cores to reduce TLB misses, and relies on three novel mechanisms to minimize address translation overheads. First, *TLB-Fill Tokens* provide *a contention-aware* mechanism to reduce thrashing in the shared L2 TLB, including a bypass cache to increase the TLB hit rate. Second, our *Address-Translation-Aware L2 Bypass* mechanism provides *contention-aware* cache bypassing to reduce interference at the L2 cache between address translation requests and data demand requests. Third, our *Address-Space-Aware DRAM Scheduler* provides a *contention-aware* memory controller policy that prioritizes address translation requests over data demand requests to mitigate high address translation overheads. Working together, these three mechanisms are highly effective at alleviating the address translation bottleneck, as our results show (Section 5).

Our comprehensive experimental evaluation shows that, via the use of TLB-request-aware policies throughout the memory hierarchy, *MASK* significantly reduces (1) the number of TLB misses that occur during multi-application execution; and (2) the overall latency of the remaining TLB misses, by ensuring that page table walks are serviced quickly. As a result, *MASK* greatly increases the average number of threads that can be scheduled during long-latency stalls, which in turn improves system throughput (weighted speedup [42, 43]) by 57.8%, improves IPC throughput by 43.4%, and reduces unfairness by 22.4% over a state-of-the-art GPU memory management unit (MMU) design [106]. *MASK* provides performance within only 23.2% of an ideal TLB that always hits.

This paper makes the following major contributions:

- To our knowledge, this is the first work to (1) provide a thorough analysis of GPU memory virtualization under multi-address-space concurrency, (2) show the large impact of address translation on latency hiding within a GPU, and (3) demonstrate the need for new techniques to alleviate contention caused by address translation due to multi-application execution in a GPU.
- We propose *MASK*, a new GPU framework that mitigates address translation overheads in the presence of multi-address-space concurrency. *MASK* consists of three novel techniques that work together to increase TLB request awareness across the entire GPU memory hierarchy. *MASK* (1) significantly improves system performance, IPC throughput, and fairness over a state-of-the-art GPU address translation mechanism; and (2) provides practical support for spatially partitioning a GPU across multiple address spaces.

2 Background

There is an increasingly pressing need to *share* the GPU hardware among multiple applications to improve GPU resource utilization. As a result, recent work [2, 17, 79, 96, 101, 102, 103] enables support for GPU virtualization, where a single physical GPU can be shared transparently across multiple applications, with each application having its own address space.[1] Much of this work relies on traditional time and spatial multiplexing techniques that are employed by CPUs, and state-of-the-art GPUs contain elements of both types of techniques [126, 130, 140]. Unfortunately, as we discuss in this section, existing GPU virtualization implementations are too coarse-grained: they employ fixed hardware policies that leave system software *without* mechanisms that can *dynamically* reallocate GPU resources to different applications, which are required for true application-transparent GPU virtualization.

2.1 Time Multiplexing

Most modern systems time-share (i.e., time multiplex) the GPU by running kernels from multiple applications back-to-back [79, 96]. These designs are optimized for the case where *no concurrency exists* between kernels from different address spaces. This simplifies memory protection and scheduling at the cost of two fundamental trade-offs. First, kernels from a single address space usually *cannot* fully utilize all of the GPU's resources, leading to significant resource underutilization [60, 69, 70, 103, 137, 138, 141]. Second, time multiplexing limits the ability of a GPU kernel scheduler to provide forward-progress or QoS guarantees, which can lead to unfairness and starvation [114].

While kernel preemption [44, 96, 101, 127, 141] could allow a time-sharing scheduler to avoid a case where one GPU kernel unfairly uses up most of the execution time (e.g., by context switching at a fine granularity), such preemption support remains an active research area in GPUs [44, 127]. Software approaches [141] sacrifice memory protection. NVIDIA's Kepler [96] and Pascal [101] architectures support preemption at the thread block and instruction granularity, respectively. We empirically find that neither granularity is effective at minimizing inter-application interference.

To illustrate the performance overhead of time multiplexing, Figure 1 shows how the execution time increases when we use time multiplexing to switch between multiple concurrently-executing processes, as opposed to executing the processes back-to-back without any concurrent execution. We perform these experiments on real NVIDIA K40 [96, 97] and NVIDIA GTX 1080 [100] GPUs. Each process runs a GPU kernel that interleaves basic arithmetic operations with loads and stores into shared and global memory. We observe that as more processes execute concurrently, the overhead of time multiplexing grows significantly. For example, on the NVIDIA GTX 1080, time multiplexing between two processes increases the total execution time by 12%, as opposed to executing one process immediately after the other process finishes. When we increase the number of processes to 10, the overhead of time multiplexing increases to 91%. On top of this high performance overhead, we find that inter-application interference pathologies (e.g., the starvation of one or more concurrently-executing application kernels) are easy to induce: an application kernel from one process consuming the majority of shared memory can easily cause application kernels from other processes to never get scheduled for execution on the GPU. While we expect preemption support to improve in future hardware, we seek a multi-application concurrency solution that does *not* depend on it.

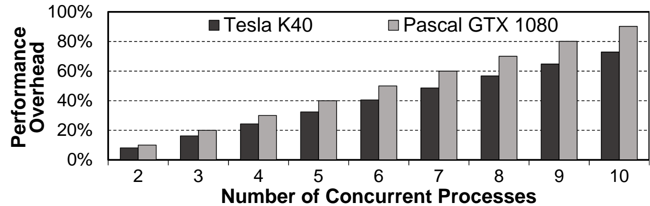


Figure 1. Increase in execution time when time multiplexing is used to execute processes concurrently on real GPUs.

2.2 Spatial Multiplexing

Resource utilization can be improved with *spatial multiplexing* [2], as the ability to execute multiple application kernels *concurrently* (1) enables the system to co-schedule kernels that have complementary resource demands, and (2) can enable independent progress guarantees for different kernels. Examples of spatial multiplexing support in modern GPUs include (1) application-specific *software* scheduling of multiple kernels [103]; and (2) NVIDIA's CUDAstream support [96, 101, 102], which co-schedules kernels from independent "streams" by merging them into a single address space. Unfortunately, these spatial multiplexing mechanisms have significant shortcomings. Software approaches (e.g., Elastic Kernels [103]) require programmers to *manually* time-slice kernels to enable their mapping onto CUDA streams for concurrency. While CUDAstream supports the flexible partitioning of resources at runtime, merging kernels into a single address space sacrifices memory protection, which is a key requirement in virtualized settings.

True GPU support for multiple concurrent address spaces can address these shortcomings by *enabling* hardware virtualization. Hardware virtualization allows the system to (1) adapt to changes in application resource utilization or (2) mitigate interference at runtime, by dynamically allocating hardware resources to multiple concurrently-executing applications. NVIDIA and AMD both offer products [6, 46] with partial hardware virtualization support. However, these products simplify memory protection by *statically* partitioning the hardware resources prior to program execution. As a result, these systems *cannot* adapt to changes in demand at runtime, and, thus, can still leave GPU resources underutilized. To efficiently support the *dynamic* sharing of GPU resources, GPUs must provide memory virtualization *and* memory protection, both of which require efficient mechanisms for virtual-to-physical address translation.

3 Baseline Design

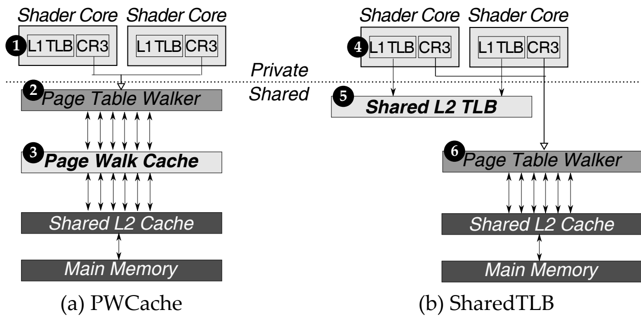
We describe (1) the state-of-the-art address translation mechanisms for GPUs, and (2) the overhead of these translation mechanisms when multiple applications share the GPU [106]. We analyze the shortcomings of state-of-the-art address translation mechanisms for GPUs in the presence of multi-application concurrency in Section 4, which motivates the need for *MASK*.

State-of-the-art GPUs extend the GPU memory hierarchy with translation lookaside buffers (TLBs) [106]. TLBs (1) greatly reduce the overhead of address translation by caching recently-used virtual-to-physical address mappings from a page table, and (2) help ensure

[1] In this paper, we use the term *address space* to refer to distinct *memory protection domains*, whose access to resources must be isolated and protected to enable GPU virtualization.

that memory accesses from application kernels running in different address spaces are isolated from each other. Recent works [105, 106] propose optimized TLB designs that improve address translation performance for GPUs.

We adopt a baseline based on these state-of-the-art TLB designs, whose memory hierarchy makes use of one of two variants for address translation: (1) *PWCache*, a previously-proposed design that utilizes a *shared page walk cache* after the L1 TLB [106] (Figure 2a); and (2) *SharedTLB*, a design that utilizes a *shared L2 TLB* after the L1 TLB (Figure 2b). The TLB caches translations that are stored in a multi-level page table (we assume a four-level page table in this work). We extend both TLB designs to handle multi-address-space concurrency. Both variants incorporate private per-core L1 TLBs, and all cores share a highly-threaded page table walker. For PWCache, on a miss in the L1 TLB (❶ in Figure 2a), the GPU initializes a page table walk (❷), which probes the shared page walk cache (❸). Any page walk requests that miss in the page walk cache go to the shared L2 cache and (if needed) main memory. For *SharedTLB*, on a miss in the L1 TLB (❹ in Figure 2b), the GPU checks whether the translation is available in the shared L2 TLB (❺). If the translation misses in the shared L2 TLB, the GPU initiates a page table walk (❻), whose requests go to the shared L2 cache and (if needed) main memory.[2]



(a) PWCache (b) SharedTLB

Figure 2. Two variants of baseline GPU design.

Figure 3 compares the performance of both baseline variants (*PWCache*, depicted in Figure 2a, and *SharedTLB*, depicted in Figure 2b), running two separate applications concurrently, to an ideal scenario where every TLB access is a hit (see Table 1 for our simulation configuration, and Section 6 for our methodology). We find that *both* variants incur a significant performance overhead (45.0% and 40.6% on average) compared to the ideal case.[3] In order to retain the benefits of sharing a GPU across multiple applications, we first analyze the shortcomings of our baseline design, and then use this analysis to develop our new mechanisms that improve TLB performance to make it approach the ideal performance.

4 Design Space Analysis

To improve the performance of address translation in GPUs, we first analyze and characterize the translation overhead in a state-of-the-art baseline (see Section 3), taking into account especially the

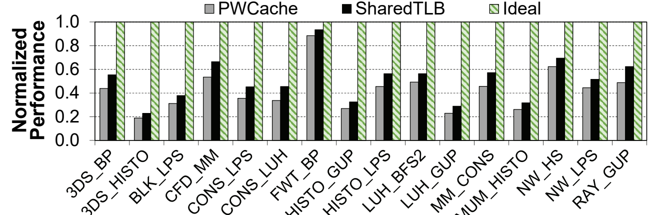


Figure 3. Baseline designs vs. ideal performance.

performance challenges induced by multi-address-space concurrency and contention. We first analyze how TLB misses can limit the GPU's ability to hide long-latency stalls, which directly impacts performance (Section 4.1). Next, we discuss two types of memory interference that impact GPU performance: (1) interference introduced by sharing GPU resources among multiple concurrent applications (Section 4.2), and (2) interference introduced by sharing the GPU memory hierarchy between address translation requests and data demand requests (Section 4.3).

4.1 Effect of TLB Misses on GPU Performance

GPU throughput relies on *fine-grained multithreading* [119, 120, 128, 129] to hide memory latency.[4] We observe a fundamental tension between address translation and fine-grained multithreading. The need to cache address translations at a page granularity, combined with application-level spatial locality, increase the likelihood that address translations fetched in response to a TLB miss are needed by *more than one* warp (i.e., many threads). Even with the massive levels of parallelism supported by GPUs, we observe that a small number of outstanding TLB misses can result in the warp scheduler not having enough ready warps to schedule, which in turn limits the GPU's essential latency-hiding mechanism.

Figure 4 illustrates a scenario for an application with four warps, where all four warps execute on the same GPU core. Figure 4a shows how the GPU behaves when no virtual-to-physical address translation is required. When Warp A performs a high-latency memory access (❶ in Figure 4), the GPU core does *not* stall since other warps have schedulable instructions (Warps B–D). In this case, the GPU core selects an active warp (Warp B) in the next cycle (❷), and continues issuing instructions. Even though Warps B–D also perform memory accesses some time later, the accesses are independent of each other, and the GPU avoids stalling by switching to a warp that is *not* waiting for a memory access (❸, ❹). Figure 4b depicts the same 4 warps *when address translation is required*. Warp A misses in the TLB (indicated in red), and stalls (❺) until the virtual-to-physical translation *finishes*. In Figure 4b, due to spatial locality within the application, the other warps (Warps B–D) need the same address translation as Warp A. As a result, they too stall (❻, ❼, ❽). At this point, the GPU no longer has any warps that it can schedule, and the GPU core stalls until the address translation request completes. Once the address translation request completes (❾), the data demand requests of the warps are issued to memory. Depending on the available memory bandwidth and the parallelism of these data demand requests, the data demand requests from Warps B–D can incur additional queuing latency (❿, ⓫, ⓬). The GPU core can resume execution *only after* the data demand request for Warp A is complete (⓭).

[2]In our evaluation, we use an 8KB page walk cache. The shared L2 TLB is located next to the shared L2 cache. L1 and L2 TLBs use the LRU replacement policy.

[3]We see discrepancies between the performance of our two baseline variants compared to the results reported by Power et al. [106]. These discrepancies occur because Power et al. assume a much higher L2 data cache access latency (130 ns vs. our 10 ns latency) and a much higher shared L2 TLB access latency (130 ns vs. our 10 ns latency). Our cache latency model, with a 10 ns access latency plus queuing latency (see Table 1 in Section 6), accurately captures modern GPU parameters [98].

[4]More detailed information about the GPU execution model and its memory hierarchy can be found in [14, 16, 17, 59, 61, 62, 63, 68, 90, 104, 111, 136, 137].

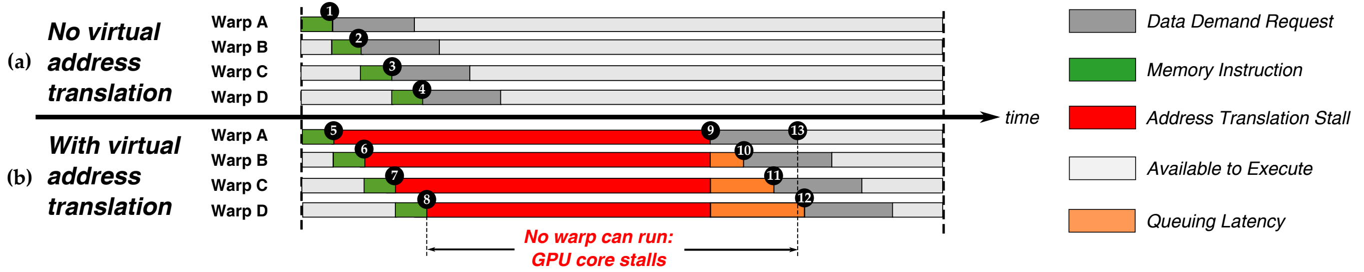


Figure 4. Example bottlenecks created by TLB misses.

Three phenomena harm performance in this scenario. First, warps stalled on TLB misses reduce the availability of schedulable warps, which lowers GPU utilization. In Figure 4, no available warp exists while the address translation request is pending, so the GPU utilization goes down to 0% for a long time. Second, address translation requests, which are a series of *dependent* memory requests generated by a page walk, must complete before a pending data demand request that requires the physical address can be issued, which reduces the GPU's ability to hide latency by keeping many memory requests in flight. Third, when the address translation data becomes available, *all* stalled warps that were waiting for the translation consecutively execute and send their data demand requests to memory, resulting in additional queuing delay for data demand requests throughout the memory hierarchy.

To illustrate how TLB misses significantly reduce the number of ready-to-schedule warps in GPU applications, Figure 5 shows the average number of concurrent page table walks (sampled every 10K cycles) for a range of applications, and Figure 6 shows the average number of stalled warps per active TLB miss, in the *SharedTLB* baseline design. Error bars indicate the minimum and maximum values. We observe from Figure 5 that more than 20 outstanding TLB misses can perform page walks at the same time, all of which contend for access to address translation structures. From Figure 6, we observe that each TLB miss can stall more than 30 warps out of the 64 warps in the core. The combined effect of these observations is that TLB misses in a GPU can quickly stall a large number of warps within a GPU core. The GPU core must wait for the misses to be resolved before issuing data demand requests and resuming execution. Hence, minimizing TLB misses and the page table walk latency is critical.

Impact of Large Pages. A large page size can significantly improve the coverage of the TLB [17]. However, a TLB miss on a large page stalls many more warps than a TLB miss on a small page. We find that with a 2MB page size, the average number of stalled warps increases to close to 100% [17], even though the average number of concurrent page table walks never exceeds 5 misses per GPU core. Regardless of the page size, there is a strong need for mechanisms that mitigate the high cost of TLB misses.

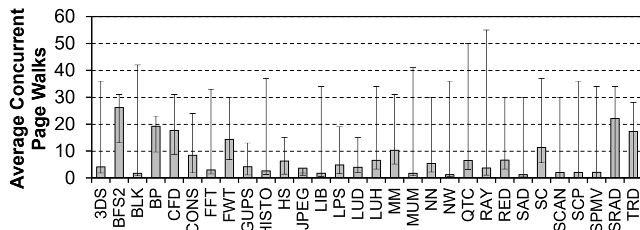


Figure 5. Average number of concurrent page walks.

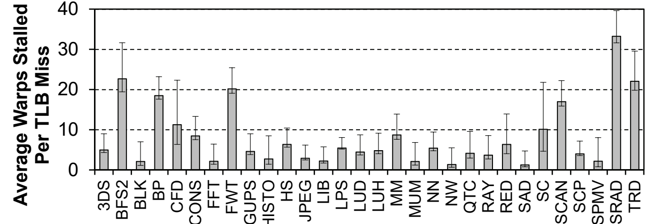


Figure 6. Average number of warps stalled per TLB miss.

4.2 Interference at the Shared TLB

When multiple applications are concurrently executed, the address translation overheads discussed in Section 4.1 are exacerbated due to inter-address-space interference. To study the impact of this interference, we measure how the TLB miss rates change once another application is introduced. Figure 7 compares the 512-entry L2 TLB miss rate of four representative workloads when each application in the workload runs in isolation to the miss rate when the two applications run concurrently and share the L2 TLB. We observe from the figure that inter-address-space interference increases the TLB miss rate significantly for most applications. This occurs because when the applications share the TLB, address translation requests often induce TLB thrashing. The resulting thrashing (1) hurts performance, and (2) leads to unfairness and starvation when applications generate TLB misses at different rates in the TLB (not shown).

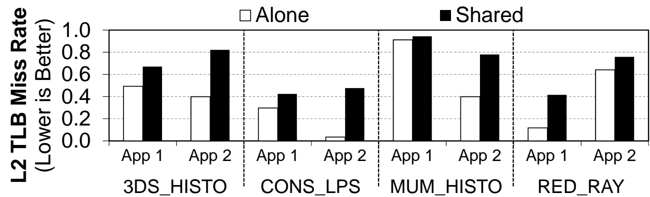


Figure 7. Effect of interference on the shared L2 TLB miss rate. Each set of bars corresponds to a pair of co-running applications (e.g., "3DS_HISTO" denotes that the 3DS and HISTO benchmarks are run concurrently).

4.3 Interference Throughout the Memory Hierarchy

Interference at the Shared Data Cache. Prior work [16] demonstrates that while cache hits in GPUs reduce the consumption of off-chip memory bandwidth, the cache hits result in a lower load/store instruction latency only when *every thread in the warp* hits in the cache. In contrast, when a page table walk hits in the shared L2 cache, the cache hit has the potential to help reduce the latency of *other warps* that have threads which access the same page in

memory. However, TLB-related data can interfere with and displace cache entries housing regular application data, which can hurt the overall GPU performance.

Hence, a trade-off exists between prioritizing address translation requests vs. data demand requests in the GPU memory hierarchy. Based on an empirical analysis of our workloads, we find that translation data from page table levels closer to the page table root are more likely to be *shared* across warps, and typically hit in the cache. We observe that, for a 4-level page table, the data cache hit rates of address translation requests across all workloads are 99.8%, 98.8%, 68.7%, and 1.0% for the root, first, second, and third levels of the page table, respectively. This means that address translation requests for the deepest page table levels often do *not* utilize the cache well. Allowing shared structures to cache page table entries from only the page table levels closer to the root could alleviate the interference between low-hit-rate address translation data and regular application data.

Interference at Main Memory. Figure 8 characterizes the DRAM bandwidth used by address translation and data demand requests, normalized to the maximum bandwidth available, for our workloads where two applications concurrently share the GPU. Figure 9 compares the average latency of address translation requests and data demand requests. We see that even though address translation requests consume only 13.8% of the total utilized DRAM bandwidth (2.4% of the maximum available bandwidth), their average DRAM latency is *higher* than that of data demand requests. This is undesirable because address translation requests usually stall multiple warps, while data demand requests usually stall only one warp (not shown). The higher latency for address translation requests is caused by the FR-FCFS memory scheduling policy [110, 152], which prioritizes accesses that hit in the row buffer. Data demand requests from GPGPU applications generally have very high row buffer locality [15, 69, 143, 150], so a scheduler that cannot distinguish address translation requests from data demand requests effectively de-prioritizes the address translation requests, increasing their latency, and thus exacerbating the effect on stalled warps.

4.4 Summary and Our Goal

We make two important observations about address translation in GPUs. First, address translation can greatly hinder a GPU's ability to hide latency by exploiting thread-level parallelism, since one single TLB miss can stall *multiple* warps. Second, during concurrent execution, multiple applications generate inter-address-space interference throughout the GPU memory hierarchy, which further increases the TLB miss latency and memory latency. In light of these observations, *our goal* is to alleviate the address translation overhead in GPUs in three ways: (1) increasing the TLB hit rate by reducing TLB thrashing, (2) decreasing interference between address translation requests and data demand requests in the shared L2 cache, and (3) decreasing the TLB miss latency by prioritizing address translation requests in DRAM without sacrificing DRAM bandwidth utilization.

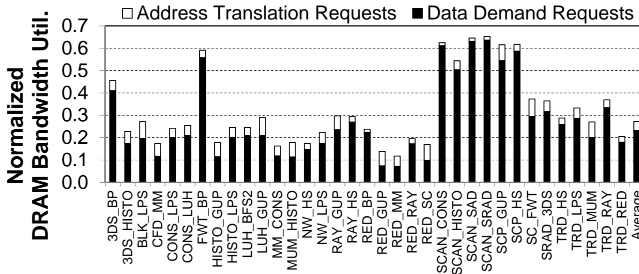


Figure 8. DRAM bandwidth utilization of address translation requests and data demand requests for two-application workloads.

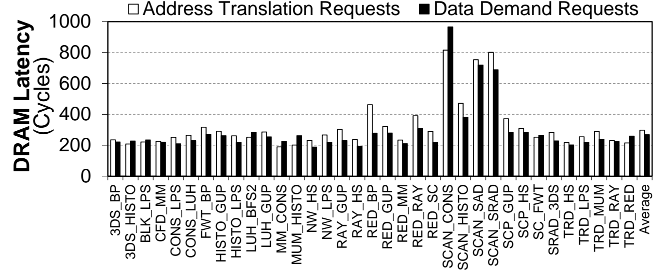


Figure 9. Latency of address translation requests and data demand requests for two-application workloads.

5 Design of MASK

To improve support for multi-application concurrency in state-of-the-art GPUs, we introduce *MASK*. *MASK* is a framework that provides memory protection support and employs three mechanisms in the memory hierarchy to reduce address translation overheads while requiring minimal hardware changes, as illustrated in Figure 10. First, we introduce *TLB-Fill Tokens*, which regulate the number of warps that can fill (i.e., insert entries) into the shared TLB in order to reduce TLB thrashing, and utilize a small TLB bypass cache to hold TLB entries from warps that are not allowed to fill the shared TLB due to not having enough tokens (❶). Second, we design an *Address-Translation-Aware L2 Bypass* mechanism, which significantly increases the shared L2 data cache utilization and hit rate by reducing interference from the TLB-related data that does not have high temporal locality (❷). Third, we design an *Address-Space-Aware DRAM Scheduler* to further reduce interference between address translation requests and data demand requests (❸). In this section, we describe the detailed design and implementation of *MASK*. We analyze the hardware cost of *MASK* in Sections 7.4 and 7.5.

5.1 Enforcing Memory Protection

Unlike previously-proposed GPU sharing techniques that do *not* enable memory protection [44, 60, 79, 96, 101, 127, 141], *MASK* provides memory protection by allowing different GPU cores to be assigned to different address spaces. *MASK* uses per-core page table *root* registers (similar to the CR3 register in x86 systems [52]) to set the current address space on each core. The page table root register value from each GPU core is also stored in a page table root cache for use by the page table walker. If a GPU core's page table root register value changes, the GPU core conservatively drains all in-flight memory requests in order to ensure correctness. We extend each L2 TLB entry with an address space identifier (ASID). TLB flush operations target a single GPU core, flushing the core's L1 TLB, and all entries in the L2 TLB that contain the matching address space identifier.

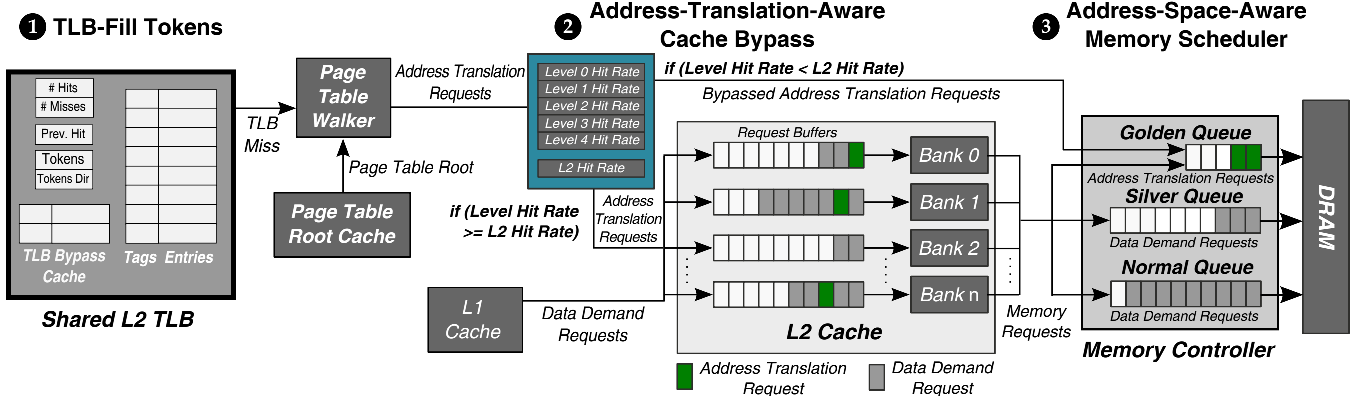


Figure 10. *MASK* design overview.

5.2 Reducing L2 TLB Interference

Sections 4.1 and 4.2 demonstrate the need to minimize TLB misses, which induce long-latency stalls. *MASK* addresses this need with a new mechanism called *TLB-Fill Tokens* (❶ in Figure 10). To reduce inter-address-space interference at the shared L2 TLB, we use an epoch- and token-based scheme to limit the number of warps from each GPU core that can fill (and therefore contend for) the L2 TLB. While every warp can probe the shared L2 TLB, only warps with tokens can *fill* the shared L2 TLB. Page table entries (PTEs) requested by warps without tokens are only buffered in a small *TLB bypass cache*. This token-based mechanism requires two components: (1) a component to determine the number of tokens allocated to each application, and (2) a component that implements a policy for assigning tokens to warps within an application.

When a TLB request arrives at the L2 TLB controller, the GPU probes tags for both the shared L2 TLB and the TLB bypass cache in parallel. A hit in either the TLB or the TLB bypass cache yields a TLB hit.

Determining the Number of Tokens. Every epoch,[5] *MASK* tracks (1) the L2 TLB miss rate for each application and (2) the total number of all warps in each core. After the first epoch,[6] the initial number of tokens for each application is set to a predetermined fraction of the total number of warps per application.

At the end of any subsequent epoch, for each application, *MASK* compares the application's shared L2 TLB miss rate during the current epoch to its miss rate from the previous epoch. If the miss rate *increases* by more than 2%, this indicates that shared TLB contention is *high* at the current token count, so *MASK decreases* the number of tokens allocated to the application. If the miss rate *decreases* by more than 2%, this indicates that shared TLB contention is *low* at the current token count, so *MASK increases* the number of tokens allocated to the application. If the miss rate change is within 2%, the TLB contention has not changed significantly, and the token count remains unchanged.

Assigning Tokens to Warps. Empirically, we observe that (1) the different warps of an application tend to have similar TLB miss rates; and (2) it is beneficial for warps that already have tokens to retain them, as it is likely that their TLB entries are already in the shared L2 TLB. We leverage these two observations to simplify the token assignment logic: our mechanism assigns tokens to warps, one token per warp, in an order based on the warp ID (i.e., if there are *n* tokens, the *n* warps with the lowest warp ID values receive tokens). This simple heuristic is effective at reducing TLB thrashing, as contention at the shared L2 TLB is reduced based on the number of tokens, and highly-used TLB entries that are requested by warps without tokens can still fill the TLB bypass cache and thus still take advantage of locality.

TLB Bypass Cache. While *TLB-Fill Tokens* can reduce thrashing in the shared L2 TLB, a handful of highly-reused PTEs may be requested by warps with no tokens, which cannot insert the PTEs into the shared L2 TLB. To address this, we add a *TLB bypass cache*, which is a small 32-entry fully-associative cache. Only warps *without* tokens can fill the TLB bypass cache in our evaluation. To preserve consistency and correctness, *MASK* flushes *all* contents of the TLB and the TLB bypass cache when a PTE is modified. Like the L1 and L2 TLBs, the TLB bypass cache uses the LRU replacement policy.

5.3 Minimizing Shared L2 Cache Interference

We find that a TLB miss generates shared L2 cache accesses with varying degrees of locality. Translating addresses through a multi-level page table (e.g., the four-level table used in *MASK*) can generate dependent memory requests at each level. This causes significant queuing latency at the shared L2 cache, corroborating observations from previous work [16]. Page table entries in levels closer to the root are more likely to be shared and thus reused across threads than entries near the leaves.

To address both interference and queuing delays due to address translation requests at the shared L2 cache, we introduce an *Address-Translation-Aware L2 Bypass* mechanism (❷ in Figure 10). To determine which address translation requests should bypass (i.e., skip probing and filling the L2 cache), we leverage our insights from Section 4.3. Recall that page table entries closer to the leaves have poor cache hit rates (i.e., the number of cache hits over all cache accesses). We make two observations from our detailed study on the page table hit rates at each page table level (see our technical report [18]). First, not all page table levels have the same hit rate across workloads (e.g., the level 3 hit rate for the MM_CONS workload is only 58.3%, but is 94.5% for RED_RAY). Second, the hit rate behavior can change over time. This means that a scheme that statically bypasses address translation requests for a certain page table level is *not* effective, as such a scheme cannot adapt to dynamic hit rate behavior changes. Because of the sharp drop-off in the L2 cache

[5]We empirically select an epoch length of 100K cycles.

[6]Note that during the first epoch, *MASK* does *not* perform TLB bypassing.

hit rate of address translation requests after the first few levels, we can simplify the mechanism to determine when address translation requests should bypass the L2 cache by comparing the L2 cache hit rate of each page table level for address translation requests to the L2 cache hit rate of data demand requests. We impose L2 cache bypassing for address translation requests from a particular page table level when the hit rate of address translation requests to that page table level falls below the hit rate of data demand requests. The shared L2 TLB has counters to track the cache hit rate of each page table level. Each memory request is tagged with a three-bit value that indicates its page walk depth, allowing *MASK* to differentiate between request types. These bits are set to *zero* for data demand requests, and to 7 for any depth higher than 6.

5.4 Minimizing Interference at Main Memory

There are two types of interference that occur at main memory: (1) data demand requests can interfere with address translation requests, as we saw in Section 4.3; and (2) data demand requests from multiple applications can interfere with each other. *MASK*'s memory controller design mitigates both forms of interference using an *Address-Space-Aware DRAM Scheduler* (❸ in Figure 10).

The *Address-Space-Aware DRAM Scheduler* breaks the traditional DRAM request buffer into three separate queues. The first queue, called the *Golden Queue*, is a small FIFO queue.[7] Address translation requests always go to the *Golden Queue*, while data demand requests go to one of the two other queues (the size of each queue is similar to the size of a typical DRAM request buffer). The second queue, called the *Silver Queue*, contains data demand requests from *one* selected application. The last queue, called the *Normal Queue*, contains data demand requests from all *other* applications. The *Golden Queue* is used to prioritize TLB misses over data demand requests. The *Silver Queue* allows the GPU to (1) avoid starvation when one or more applications hog memory bandwidth, and (2) improve fairness when multiple applications execute concurrently [15, 84]. When one application unfairly hogs DRAM bandwidth in the *Normal Queue*, the *Silver Queue* can process data demand requests from another application that would otherwise be starved or unfairly delayed.

Our *Address-Space-Aware DRAM Scheduler* always prioritizes requests in the *Golden Queue* over requests in the *Silver Queue*, which are always prioritized over requests in the *Normal Queue*. To provide higher priority to applications that are likely to be stalled due to concurrent TLB misses, and to minimize the time that bandwidth-heavy applications have access to the silver queue, each application takes turns being assigned to the *Silver Queue* based on two per-application metrics: (1) the number of concurrent page walks, and (2) the number of warps stalled per active TLB miss. The number of data demand requests each application can add to the *Silver Queue*, when the application gets its turn, is shown as $thresh_i$ in Equation 1. After application i (App_i) reaches its quota, the next application (App_{i+1}) is then allowed to send its requests to the *Silver Queue*, and so on. Within both the *Silver Queue* and *Normal Queue*, FR-FCFS [110, 152] is used to schedule requests.

$$thresh_i = thresh_{max} x \frac{ConPTW_i * WarpsStalled_i}{\sum_{j=1}^{numApps} ConPTW_j * WarpsStalled_j} \quad (1)$$

[7]We observe that address translation requests have low row buffer locality. Thus, there is no significant performance benefit if the memory controller reorders address translation requests within the *Golden Queue* to exploit row buffer locality.

To track the number of outstanding concurrent page walks (*ConPTW* in Equation 1), we add a 6-bit counter per application to the shared L2 TLB.[8] This counter tracks the number of concurrent TLB misses. To track the number of warps stalled per active TLB miss (*WarpsStalled* in Equation 1), we add a 6-bit counter to each TLB MSHR entry, which tracks the maximum number of warps that hit in the entry. The *Address-Space-Aware DRAM Scheduler* resets all of these counters every epoch (see Section 5.2).

We find that the number of concurrent address translation requests that go to each memory channel is small, so our design has an additional benefit of lowering the page table walk latency (because it prioritizes address translation requests) while minimizing interference.

5.5 Page Faults and TLB Shootdowns

Address translation inevitably introduces page faults. Our design can be extended to use techniques from previous works, such as performing copy-on-write for handling page faults [106], and either exception support [83] or demand paging techniques [10, 101, 151] for major faults. We leave this as future work.

Similarly, TLB shootdowns are required when a GPU core changes its address space or when a page table entry is updated. Techniques to reduce TLB shootdown overhead [26, 113, 149] are well-explored and can be used with *MASK*.

6 Methodology

To evaluate *MASK*, we model the NVIDIA Maxwell architecture [98], and the TLB-fill bypassing, cache bypassing, and memory scheduling mechanisms in *MASK*, using the Mosaic simulator [17], which is based on GPGPU-Sim 3.2.2 [20]. We heavily modify the simulator to accurately model the behavior of CUDA Unified Virtual Addressing [98, 101] as described below. Table 1 provides the details of our baseline GPU configuration. Our baseline uses the FR-FCFS memory scheduling policy [110, 152], based on findings from previous works [15, 29, 150] which show that FR-FCFS provides good performance for GPGPU applications compared to other, more sophisticated schedulers [71, 72]. We have open-sourced our modified simulator online [115].

GPU Core Configuration	
System Overview	30 cores, 64 execution units per core.
Shader Core	1020 MHz, 9-stage pipeline, 64 threads per warp, GTO scheduler [112].
Page Table Walker	Shared page table walker, traversing 4-level page tables.
Cache and Memory Configuration	
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before accessing L2, 1-cycle latency.
Private L1 TLB	64 entries per core, fully associative, LRU, 1-cycle latency.
Shared L2 Cache	2MB total, 16-way associative, LRU, 16 cache banks, 2 ports per cache bank, 10-cycle latency
Shared L2 TLB	512 entries total, 16-way associative, LRU, 2 ports, 10-cycle latency
Page Walk Cache	16-way 8KB, 10-cycle latency
DRAM	GDDR5 1674 MHz [118], 8 channels, 8 banks per rank, 1 rank, FR-FCFS scheduler [110, 152], burst length 8

Table 1. Configuration of the simulated system.

[8]We leave techniques to virtualize this counter for more than 64 applications as future work.

TLB and Page Table Walker Model. We accurately model both TLB design variants discussed in Section 3. We employ the non-blocking TLB implementation used by Pichai et al. [105]. Each core has a private L1 TLB. The page table walker is shared across threads, and admits up to 64 concurrent threads for walks. On a TLB miss, a page table walker generates a series of dependent requests that probe the L2 cache and main memory as needed. We faithfully model the multi-level page walks.

Workloads. We randomly select 27 applications from the CUDA SDK [94], Rodinia [31], Parboil [121], LULESH [65, 66], and SHOC [37] suites. We classify these benchmarks based on their L1 and L2 TLB miss rates into one of four groups, as shown in Table 2. For our multi-application results, we randomly select 35 pairs of applications, avoiding pairs where both applications have a low L1 TLB miss rate (i.e., <20%) and low L2 TLB miss rate (i.e., <20%), since these applications are relatively insensitive to address translation overheads. The application that finishes first is relaunched to keep the GPU core busy and maintain memory contention.

L1 TLB Miss Rate	L2 TLB Miss Rate	Benchmark Name
Low	Low	LUD, NN
Low	High	BFS2, FFT, HISTO, NW, QTC, RAY, SAD, SCP
High	Low	BP, GUP, HS, LPS
High	High	3DS, BLK, CFD, CONS, FWT, LUH, MM, MUM, RED, SC, SCAN, SRAD, TRD

Table 2. Categorization of workloads.

We divide 35 application-pairs into three workload categories based on the number of applications that have both high L1 and L2 TLB miss rates, as high TLB miss rates at both levels indicate a high amount of pressure on the limited TLB resources. *n-HMR* contains application-pairs where *n* applications in the workload have both high L1 and L2 TLB miss rates.

Evaluation Metrics. We report performance using *weighted speedup* [42, 43], a commonly-used metric to evaluate the performance of a multi-application workload [15, 38, 39, 68, 71, 72, 86, 88, 89, 124, 125, 132]. Weighted speedup is defined as $\sum \frac{IPC_{Shared}}{IPC_{Alone}}$, where IPC_{alone} is the IPC of an application that runs on the same number of GPU cores, but does *not* share GPU resources with any other application, and IPC_{shared} is the IPC of an application when it runs concurrently with other applications. We report the unfairness of each design using *maximum slowdown*, defined as $Max \frac{IPC_{Alone}}{IPC_{Shared}}$ [15, 38, 41, 71, 72, 122, 123, 124, 125, 132, 133].

Scheduling and Partitioning of Cores. We assume an oracle GPU scheduler that finds the *best* partitioning of the GPU cores for each pair of applications. For each pair of applications that are concurrently executed, the scheduler partitions the cores according to the best weighted speedup for that pair found by an exhaustive search over all possible static core partitionings. *Neither the L2 cache nor main memory are partitioned.* All applications can use all of the shared L2 cache and the main memory.

Design Parameters. *MASK* exposes two configurable parameters: *InitialTokens* for *TLB-Fill Tokens*, and $thresh_{max}$ for the *Address-Space-Aware DRAM Scheduler*. A sweep over the range of possible *InitialTokens* values reveals less than 1% performance variance, as *TLB-Fill Tokens* are effective at reconfiguring the total number of tokens to a steady-state value (Section 5.2). In our evaluation, we set *InitialTokens* to 80%. We set $thresh_{max}$ to 500 empirically.

7 Evaluation

We compare the performance of *MASK* against four GPU designs. The first, called *Static*, uses a static spatial partitioning of resources, where an oracle is used to partition GPU cores, but the shared L2 cache and memory channels are partitioned equally across applications. This design is intended to capture key design aspects of NVIDIA GRID [46] and AMD FirePro [6], based on publicly-available information. The second design, called *PWCache*, models the page walk cache baseline design we discuss in Section 3. The third design, called *SharedTLB*, models the shared L2 TLB baseline design we discuss in Section 3. The fourth design, *Ideal*, represents a hypothetical GPU where every single TLB access is a TLB hit. In addition to these designs, we report the performance of the individual components of *MASK*: *TLB-Fill Tokens* (*MASK*-TLB), *Address-Translation-Aware L2 Bypass* (*MASK*-Cache), and *Address-Space-Aware DRAM Scheduler* (*MASK*-DRAM).

7.1 Multiprogrammed Performance

Figure 11 compares the average performance by workload category of *Static*, *PWCache*, *SharedTLB*, and *Ideal* to *MASK* and the three individual components of *MASK*. We make two observations from Figure 11. First, compared to *SharedTLB*, which is the best-performing baseline, *MASK* improves the weighted speedup by 57.8% on average. Second, we find that *MASK* performs only 23.2% worse than *Ideal* (where all accesses to the L1 TLB are hits). This demonstrates that *MASK* reduces a large portion of the TLB miss overhead.

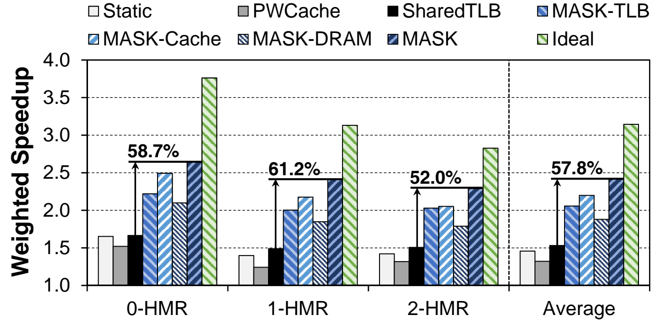


Figure 11. Multiprogrammed workload performance, grouped by workload category.

Individual Workload Performance. Figures 12, 13, and 14 compare the weighted speedup of each individual multiprogrammed workload for *MASK*, and the individual performance of its three components (*MASK*-TLB, *MASK*-Cache, and *MASK*-DRAM), against *Static*, *PWCache*, and *SharedTLB* for the 0-HMR (Figure 12), 1-HMR (Figure 13), and 2-HMR (Figure 14) workload categories. Each group of bars in Figures 12–14 represents a pair of co-scheduled benchmarks. We make two observations from the figures. First, compared to *Static*, where resources are statically partitioned, *MASK* provides better performance, because when an application stalls for concurrent TLB misses, it no longer needs a large amount of other shared resources, such as DRAM bandwidth. During such stalls, other applications can utilize these resources. When multiple GPGPU applications run concurrently using *MASK*, TLB misses from two or more applications can be staggered, increasing the likelihood that there will be heterogeneous and complementary resource demands. Second, *MASK* provides significant performance

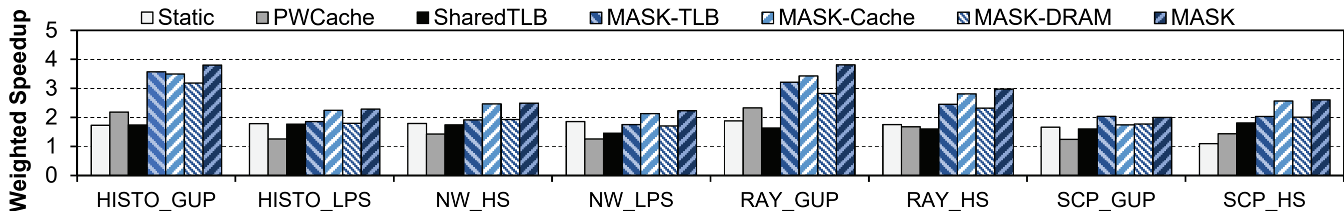


Figure 12. Performance of multiprogrammed workloads in the 0-HMR workload category.

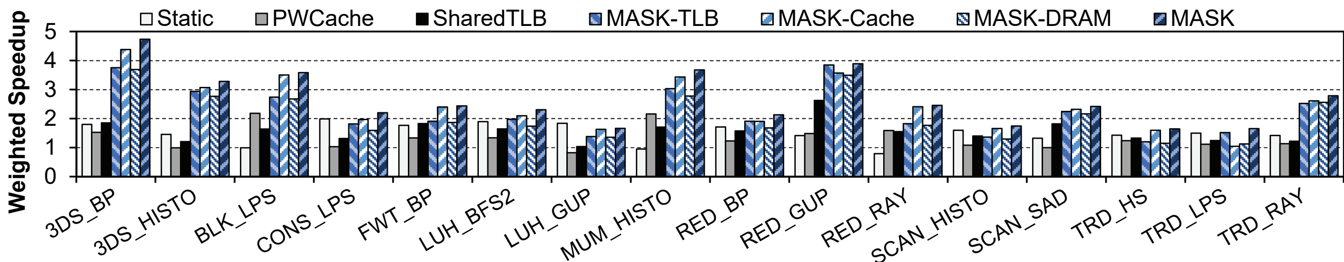


Figure 13. Performance of multiprogrammed workloads in the 1-HMR workload category.

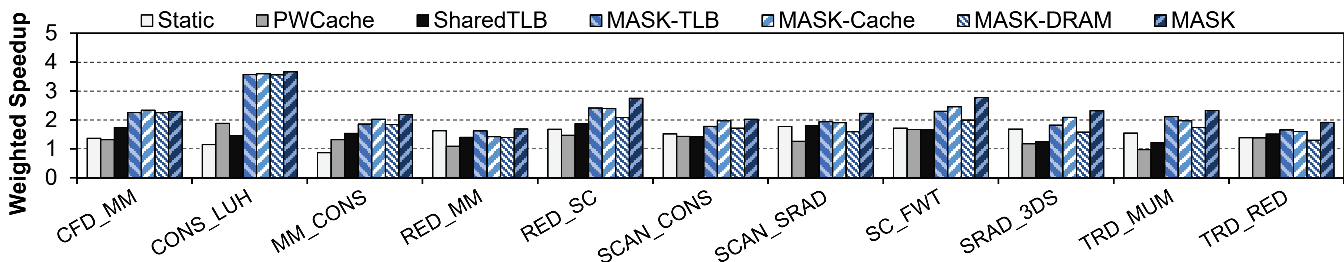


Figure 14. Performance of multiprogrammed workloads in the 2-HMR workload category.

improvements over *both PWCache* and *SharedTLB* regardless of the workload type (i.e., 0-HMR to 2-HMR). This indicates that *MASK* is effective at reducing the address translation overhead both when TLB contention is high and when TLB contention is relatively low.

Our technical report [18] provides additional analysis on the aggregate throughput (system-wide IPC). In the report, we show that *MASK* provides 43.4% better aggregate throughput compared to *SharedTLB*.

Figure 15 compares the unfairness of *MASK* to that of *Static*, *PWCache*, and *SharedTLB*. We make two observations. First, compared to statically partitioning resources (*Static*), *MASK* provides better fairness by allowing *both* applications to access all shared resources. Second, compared to *SharedTLB*, which is the baseline that provides the best fairness, *MASK* reduces unfairness by 22.4% on average. As the number of tokens for each application changes based on the L2 TLB miss rate, applications that benefit more from the shared L2 TLB are more likely to get more tokens, causing applications that do not benefit from shared L2 TLB space to yield that shared L2 TLB space to other applications. Our application-aware token distribution mechanism and TLB-fill bypassing mechanism work in tandem to reduce the amount of shared L2 TLB thrashing observed in Section 4.2.

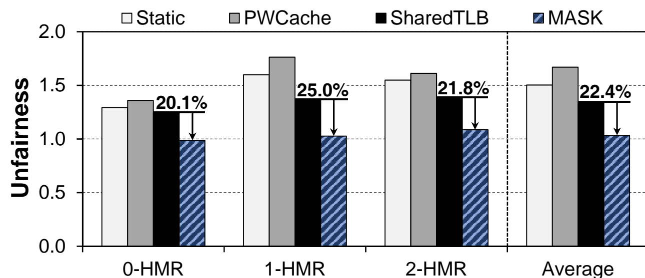


Figure 15. Multiprogrammed workload unfairness.

Individual Application Analysis. *MASK* provides better throughput for *all* individual applications sharing the GPU due to reduced TLB miss rates for each application (shown in our technical report [18]). The per-application L2 TLB miss rates are reduced by over 50% on average, which is in line with the reduction in system-wide L2 TLB miss rates (see Section 7.2). Reducing the number of TLB misses via our TLB-fill bypassing policy (Section 5.2), and reducing the latency of TLB misses via our shared L2 bypassing (Section 5.3) and TLB- and application-aware DRAM scheduling (Section 5.4) policies, enables significant performance improvement.

In some cases, running two applications concurrently provides *better* performance as well as lower unfairness than running each application alone (e.g., for the RED_BP and RED_RAY workloads in Figure 13, and the SC_FWT workload in Figure 14). We attribute such cases to substantial improvements (more than 10%) of two factors: a lower L2 cache queuing latency for bypassed address translation requests, and a higher L1 cache hit rate of data demand requests when applications share the L2 cache and main memory with other applications.

We conclude that *MASK* is effective at reducing the address translation overheads in modern GPUs, and thus at improving both performance and fairness, by introducing address translation request awareness throughout the GPU memory hierarchy.

7.2 Component-by-Component Analysis

This section characterizes *MASK*'s underlying mechanisms (*MASK*-TLB, *MASK*-Cache, and *MASK*-DRAM). Figure 11 shows the average performance improvement of each individual component of *MASK* compared to *Static*, *PWCache*, *SharedTLB*, and *MASK*. We summarize our key findings here, and provide a more detailed analysis in our technical report [18].

Effectiveness of *TLB-Fill Tokens.* *MASK* uses *TLB-Fill Tokens* to reduce thrashing. We compare TLB hit rates for *Static*, *SharedTLB*, and *MASK*-TLB. The hit rates for *Static* and *SharedTLB* are substantially similar. *MASK*-TLB increases shared L2 TLB hit rates by 49.9% on average over *SharedTLB* [18], because the *TLB-Fill Tokens* mechanism reduces the number of warps utilizing the shared L2 TLB entries, in turn reducing the miss rate. The TLB bypass cache stores frequently-used TLB entries that cannot be filled in the traditional TLB. Measurement of the average TLB bypass cache hit rate (66.5%) confirms this conclusion [18].[9]

Effectiveness of *Address-Translation-Aware L2 Bypass.* *MASK* uses *Address-Translation-Aware L2 Bypass* with the goal of prioritizing address translation requests. We measure the average L2 cache hit rate for address translation requests. We find that for address translation requests that fill into the shared L2 cache, *Address-Translation-Aware L2 Bypass* is very effective at selecting which blocks to cache, resulting in an address translation request hit rate that is higher than 99% for all of our workloads. At the same time, *Address-Translation-Aware L2 Bypass* minimizes the impact of long L2 cache queuing latency [16], leading to a 43.6% performance improvement compared to *SharedTLB* (as shown in Figure 11).

Effectiveness of *Address-Space-Aware DRAM Scheduler.* To characterize the performance impact of *MASK*'s DRAM scheduler, we compare the DRAM bandwidth utilization and average DRAM latency of (1) address translation requests and (2) data demand requests for the baseline designs and *MASK*, and make two observations. First, we find that *MASK* is effective at reducing the DRAM latency of address translation requests, which contributes to the 22.7% performance improvement of *MASK*-DRAM over *SharedTLB*, as shown in Figure 11. In cases where the DRAM latency is high, our DRAM scheduling policy reduces the latency of address translation requests by up to 10.6% (SCAN_SAD), while increasing DRAM bandwidth utilization by up to 5.6% (SCAN_HISTO). Second, we find that when an application is suffering severely from interference due to another concurrently-executing application, the *Silver Queue* significantly reduces the latency of data demand requests from the suffering application. For example, when the *Silver Queue* is employed, SRAD from the SCAN_SRAD application-pair performs 18.7% better, while both SCAN and CONS from SCAN_CONS perform 8.9% and 30.2% better, respectively. Our technical report [18] provides a more detailed analysis of the impact of our *Address-Space-Aware DRAM Scheduler*.

We conclude that each component of *MASK* provides complementary performance improvements by introducing address-translation-aware policies at different memory hierarchy levels.

7.3 Scalability and Generality

This section evaluates the scalability of *MASK* and provides evidence that the design generalizes well across different architectures. We summarize our key findings here, and provide a more detailed analysis in our technical report [18].

Scalability. We compare the performance of *SharedTLB*, which is the best-performing state-of-the-art baseline design, and *MASK*, normalized to *Ideal* performance, as the number of concurrently-running applications increases from one to five. In general, as the application count increases, contention for shared resources (e.g., shared L2 TLB, shared L2 cache) draws the performance for both *SharedTLB* and *MASK* further from the performance of *Ideal*. However, *MASK* maintains a consistent performance advantage relative to *SharedTLB*, as shown in Table 3. The performance gain of *MASK* relative to *SharedTLB* is more pronounced at higher levels of multi-application concurrency because (1) the shared L2 TLB becomes heavily contended as the number of concurrent applications increases, and (2) *MASK* is effective at reducing the amount of contention at the heavily-contended shared TLB.

Number of Applications	1	2	3	4	5
SharedTLB performance normalized to *Ideal*	47.1%	48.7%	38.8%	34.2%	33.1%
MASK performance normalized to *Ideal*	68.5%	76.8%	62.3%	55.0%	52.9%

Table 3. Normalized performance of *SharedTLB* and *MASK* as the number of concurrently-executing applications increases.

Generality. *MASK* is an architecture-independent design: our techniques are applicable to any SIMT machine [4, 5, 12, 95, 96, 98, 101, 107, 139]. To demonstrate this, we evaluate our two baseline variants (*PWCache* and *SharedTLB*) and *MASK* on two additional GPU architectures: the GTX480 (Fermi architecture [95]), and an integrated GPU architecture [3, 12, 27, 34, 49, 53, 92, 93, 106, 107, 142], as shown in Table 4. We make three key conclusions. First, address translation leads to significant performance overhead in both *PWCache* and *SharedTLB*. Second, *MASK* provides a 46.9% average performance improvement over *PWCache* and a 29.1% average performance improvement over *SharedTLB* on the Fermi architecture, getting to within 22% of the performance of *Ideal*. Third, on the integrated GPU configuration used in previous work [106], we find that *MASK* provides a 23.8% performance improvement over *PWCache* and a 68.8% performance improvement over *SharedTLB*, and gets within 35.5% of the performance of *Ideal*.

Relative Performance	Fermi	Integrated GPU [106]
PWCache	53.1%	52.1%
SharedTLB	60.4%	38.2%
MASK	78.0%	64.5%

Table 4. Average performance of *PWCache*, *SharedTLB*, and *MASK*, normalized to *Ideal*.

We conclude that *MASK* is effective at (1) reducing the performance overhead of address translation, and (2) significantly improving system performance over *both* the *PWCache* and *SharedTLB* designs, regardless of the GPU architecture.

Sensitivity to L1 and L2 TLB Sizes. We evaluate the benefit of MASK over many different TLB sizes in our technical report [18]. We make two observations. First, *MASK* is effective at reducing (1) TLB thrashing at the shared L2 TLB, and (2) the latency of address translation requests regardless of TLB size. Second, as we increase the shared L2 TLB size from 64 to 8192 entries, *MASK* outperforms *SharedTLB* for all TLB sizes except the 8192-entry shared L2 TLB. At 8192 entries, *MASK* and *SharedTLB* perform equally, because the working set fits completely within the 8192-entry shared L2 TLB.

[9] We find that the performance of *MASK*-TLB saturates when we increase the TLB bypass cache beyond 32 entries for the workloads that we evaluate.

Sensitivity to Memory Policies. We study the sensitivity of *MASK* to (1) main memory row policy, and (2) memory scheduling policies. We find that for all of our baselines and for *MASK*, performance with an open-row policy [71] is similar (within 0.8%) to the performance with a closed-row policy, which is used in various CPUs [47, 48, 53]. Aside from the FR-FCFS scheduler [110, 152], we use *MASK* in conjunction with another state-of-the-art GPU memory scheduler [60], and find that with this scheduler, *MASK* improves performance by 44.2% over *SharedTLB*. We conclude that *MASK* is effective across different memory policies.

Sensitivity to Different Page Sizes. We evaluate the performance of *MASK* with 2MB large pages assuming an ideal page fault latency [14, 18] (not shown). We provide two observations. First, even with the larger page size, *SharedTLB* continues to experience high contention during address translation, causing its average performance to fall 44.5% short of Ideal. Second, we find that using *MASK* allows the GPU to perform within 1.8% of Ideal.

7.4 Storage Cost

To support memory protection, each L2 TLB entry has an 9-bit address space identifier (ASID), which translates to an overhead of 7% of the L2 TLB size in total.

At each core, our *TLB-Fill Tokens* mechanism uses (1) two 16-bit counters to track the shared L2 TLB hit rate, with one counter tracking the number of shared L2 TLB hits, and the other counter tracking the number of shared L2 TLB misses; (2) a 256-bit vector addressable by warp ID to track the number of active warps, where each bit is set when a warp uses the shader core for the first time, and is reset every epoch; and (3) an 8-bit incrementer that tracks the total number of unique warps executed by the core (i.e., its counter value is incremented each time a bit is set in the bit vector).

We augment the shared cache with a 32-entry fully-associative content addressable memory (CAM) for the bypass cache, 30 15-bit token counters, and 30 1-bit direction registers to record whether the token count increased or decreased during the previous epoch. These structures allow the GPU to distribute tokens among up to 30 concurrent applications. In total, we add 706 bytes of storage (13 bytes per core in the L1 TLB, and 316 bytes total in the shared L2 TLB), which adds 1.6% to the baseline L1 TLB size and 3.8% to the baseline L2 TLB size (in addition to the 7% overhead due to the ASID bits).

Address-Translation-Aware L2 Bypass uses ten 8-byte counters per core to track L2 cache hits and L2 cache accesses per level. The resulting 80 bytes add less than 0.1% to the baseline shared L2 cache size. Each L2 cache and memory request requires an additional 3 bits to specify the page walk level, as we discuss in Section 5.3.

For each memory channel, our *Address-Space-Aware DRAM Scheduler* contains a 16-entry FIFO queue for the *Golden Queue*, a 64-entry memory request buffer for the *Silver Queue*, and a 192-entry memory request buffer for the *Normal Queue*. This adds an extra 6% of storage overhead to the DRAM request queue per memory controller.

7.5 Chip Area and Power Consumption

We compare the area and power consumption of *MASK* to *PWCache* and *SharedTLB* using CACTI [87]. *PWCache* and *SharedTLB* have near-identical area and power consumption, as we size the page walk cache and shared L2 TLB (see Section 3) such that they both use the same total area. We find that *MASK* introduces a negligible overhead to both baselines, consuming less than 0.1% additional area and 0.01% additional power in each baseline. We provide a detailed analysis of area and power consumption in our technical report [18].

8 Related Work

To our knowledge, this paper is the first to (1) provide a thorough analysis of GPU memory virtualization under multi-application concurrency, and (2) redesign the entire GPU memory hierarchy to be aware of address translation requests. In this section, we discuss previous techniques that aim to (1) provide sharing and virtualization mechanisms for GPUs, (2) reduce the overhead of address translation in GPUs, and (3) reduce inter-application interference.

8.1 Techniques to Enable GPU Sharing

Spatial Multiplexing and Multi-Application Concurrency. Several works propose techniques to improve GPU utilization with multi-application concurrency [17, 60, 78, 103, 141, 148], but they do not support memory protection. Jog et al. [60] propose an application-aware GPU memory scheduler to improve the performance of concurrently-running GPU applications. Adriaens et al. [2] observe the need for spatial sharing across protection domains, but do not propose or evaluate a design. NVIDIA GRID [46] and AMD Firepro [6] support static partitioning of hardware to allow kernels from different VMs to run concurrently, but the partitions are determined at startup, which causes fragmentation and underutilization (see the *Static* configuration evaluated in Section 7.1). *MASK*'s goal is *flexible, dynamic* partitioning. NVIDIA's Multi Process Service (MPS) [99] allows multiple processes to launch kernels on the GPU, but the service provides no memory protection or error containment. Xu et al. [147] propose Warped-Slicer, which is a mechanism for multiple applications to spatially share a GPU core. Warped-Slicer provides no memory protection, and is not suitable for supporting multi-application execution. Ausavarungnirun et al. [17] propose Mosaic, a mechanism that provides programmer-transparent support for multiple page sizes in GPUs that provide memory protection for multi-application execution. The main goal of Mosaic is to increase the effective size of the TLB, which is orthogonal to *MASK*, and Mosaic can be combined with *MASK* to achieve even higher performance, as shown in [14].

Time Multiplexing. As discussed in Section 2.1, time multiplexing is an active research area [44, 127, 141], and architectural support [79, 101] will likely improve in future GPUs. These techniques are complementary to and can be combined with *MASK*.

GPU Virtualization. Techniques to provide GPU virtualization can support concurrent execution of GPGPU applications [67, 126, 130]. However, these GPU virtualization approaches require dedicated hardware support, such as the Virtual Desktop Infrastructure (VDI) found in NVIDIA GRID [46] and AMD FirePro [6]. As we discuss in Section 2.2, these techniques do *not* provide *dynamic* partitioning of hardware resources, and perform worse than *MASK* (see Section 7). vmCUDA [140] and rCUDA [40] provide close-to-ideal performance, but they require significant modifications to GPGPU applications and the operating system, which sacrifice transparency to the application, performance isolation, and compatibility across multiple GPU architectures. Vijaykumar et al. [136]

propose a framework to virtualize GPU resources for a single application. This technique is complementary to *MASK*.

Demand Paging. Recent works on CC-NUMA [10], AMD's hUMA [8], and NVIDIA's PASCAL architecture [101, 151] support demand paging in GPUs. These techniques can be used in conjunction with *MASK*.

8.2 TLB Design

GPU TLB Designs. Previous works [17, 35, 105, 106, 134] explore TLB designs in heterogeneous systems with GPUs. Cong et al. [35] propose a TLB design for accelerators. This design utilizes the host (CPU) MMU to perform page walks, which results in high performance overhead in the context of multi-application GPUs. Pichai et al. [105] explore a TLB design for heterogeneous CPU-GPU systems, and add TLB awareness to the existing CCWS GPU warp scheduler [112]. Warp scheduling is orthogonal to our work, and can be combined to further improve performance.

Vesely et al. [134] analyze support for virtual memory in heterogeneous systems, finding that the cost of address translation in GPUs is an order of magnitude higher than that in CPUs, and that high-latency address translations limit the GPU's latency hiding capability and hurt performance. We show additionally that inter-address-space interference further slows down applications sharing the GPU. *MASK* is capable of not only reducing interference between multiple applications (Section 7.1), but also reducing the TLB miss rate during single-application execution as well (as shown in our technical report [18]).

Lee et al. [75] propose VAST, a software runtime that dynamically partitions GPU kernels to manage the memory available to each kernel. VAST does not provide any support for memory protection. In contrast, *MASK* enables memory virtualization in hardware, and offers memory protection.

TLB Designs in CPU Systems. Cox and Bhattacharjee [36] propose a TLB design that allows entries corresponding to multiple page sizes to share the same TLB structure, simplifying the design of TLBs. This design solves a different problem (area and energy efficiency), and is orthogonal to *MASK*. Other works [24, 25, 81] examine shared last-level TLB designs and page walk cache designs [23], proposing mechanisms that can accelerate *multithreaded* applications by sharing translations between cores. These proposals are likely to be less effective for *multiple* concurrent GPGPU applications, for which translations are *not* shared between virtual address spaces. Barr et al. [21] propose SpecTLB, which speculatively predicts address translations to avoid the TLB miss latency. Doing so can be costly in GPUs, because there can be multiple concurrent TLB misses to many different TLB entries in the GPU.

Direct segments [22] and redundant memory mappings [64] reduce address translation overheads by mapping large contiguous virtual memory regions to a contiguous physical region. These techniques increase the reach of each TLB entry, and are complementary to *MASK*.

8.3 Techniques to Reduce Inter-Application Interference

GPU-Specific Resource Management. Lee et al. [74] propose TAP, a TLP-aware cache management mechanism that modifies the utility-based cache partitioning policy [109] and the baseline cache insertion policy [56] to lower cache space interference between GPGPU and CPU applications. TAP does not consider address translation traffic.

Various memory scheduler designs target systems with GPUs [15, 30, 57, 63, 131, 132, 150]. Unlike *MASK*, these designs focus on a single GPGPU application, and are not aware of address translation requests. While some of these works propose mechanisms that reduce inter-application interference [15, 57, 131, 132], they differ from *MASK* because they (1) consider interference bewteen CPU applications and GPU applications, and (2) are *not* aware of address translation requests.

Cache Bypassing Policies in GPUs. There are many techniques (e.g., [16, 32, 33, 58, 76, 77, 145, 146]) to reduce contention in shared GPU caches. Unlike *MASK*, these works do *not* differentiate address translation requests from data demand requests, and focus on only single-application execution.

Cache and TLB Insertion Policies. Cache insertion policies that account for cache thrashing [55, 56, 108], future reuse [116, 117], or inter-application interference [144] work well for CPU applications, but previous works have shown that some of these policies can be ineffective for GPU applications [16, 74]. This observation holds for the shared L2 TLB in multi-address space execution.

9 Conclusion

Spatial multiplexing support, which allows multiple applications to run concurrently, is needed to efficiently deploy GPUs in a large-scale computing environment. Unfortunately, due to the primitive existing support for memory virtualization, many of the performance benefits of spatial multiplexing are lost in state-of-the-art GPUs. We perform a detailed analysis of state-of-the-art mechanisms for memory virtualization, and find that current address translation mechanisms (1) are highly susceptible to interference across the different address spaces of applications in the shared TLB structures, which leads to a high number of page table walks; and (2) undermine the fundamental latency-hiding techniques of GPUs, by often stalling hundreds of threads at once. To alleviate these problems, we propose *MASK*, a new memory hierarchy designed carefully to support multi-application concurrency at low overhead. *MASK* consists of three major components in different parts of the memory hierarchy, all of which incorporate address translation request awareness. These three components work together to lower inter-application interference during address translation, and improve L2 cache utilization and memory latency for address translation requests. *MASK* improves performance by 57.8%, on average across a wide range of multiprogrammed workloads, over the state-of-the-art. We conclude that *MASK* provides a promising and effective substrate for multi-application execution on GPUs, and hope future work builds on the mechanism we provide and open source [115].

Acknowledgments

We thank the anonymous reviewers from ASPLOS 2017/2018, MICRO 2016/2017, and ISCA 2017. We gratefully acknowledge SAFARI and SCEA group members for their feedback. We acknowledge the support of our industrial partners, especially Google, Intel, Microsoft, NVIDIA, and VMware. This research was partially supported by the NSF (grants 1409723, 1618563, 1657336, and 1750667) and the Semiconductor Research Corporation. An earlier version of this paper was placed on arXiv.org in August 2017 [19].

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," http://download.tensorflow.org/paper/whitepaper2015.pdf, 2015.

[2] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The Case for GPGPU Spatial Multitasking," in *HPCA*, 2012.

[3] Advanced Micro Devices, Inc., "AMD Accelerated Processing Units," http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx.

[4] Advanced Micro Devices, Inc., "AMD Radeon R9 290X," http://www.amd.com/us/press-releases/Pages/amd-radeon-r9-290x-2013oct24.aspx.

[5] Advanced Micro Devices, Inc., "ATI Radeon GPGPUs," http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/Pages/amd-radeon-hd-6000.aspx.

[6] Advanced Micro Devices, Inc., "OpenCL: The Future of Accelerated Application Performance Is Now," https://www.amd.com/Documents/FirePro_OpenCL_Whitepaper.pdf.

[7] Advanced Micro Devices, Inc., *AMD-V Nested Paging*, 2010, http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[8] Advanced Micro Devices, Inc., "Heterogeneous System Architecture: A Technical Review," http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf, 2012.

[9] Advanced Micro Devices, Inc., "AMD I/O Virtualization Technology (IOMMU) Specification," http://support.amd.com/TechDocs/48882_IOMMU.pdf, 2016.

[10] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *HPCA*, 2015.

[11] J. B. Alex Chen and X. Amatriain, "Distributed Neural Networks with GPUs in the AWS Cloud," http://techblog.netflix.com/2014/02/distributed-neural-networks-with-gpus.html, 2014.

[12] ARM Holdings PLC, "Take GPU Processing Power Beyond Graphics with Mali GPU Computing," 2012.

[13] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, and C.-J. Wu, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *ISCA*, 2017.

[14] R. Ausavarungnirun, "Techniques for Shared Resource Management in Systems with Throughput Processors," Ph.D. dissertation, Carnegie Mellon Univ., 2017.

[15] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[16] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *PACT*, 2015.

[17] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes," in *MICRO*, 2017.

[18] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Spatial Multiplexing Support for Multi-Application Concurrency in GPUs," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. TR-2018-002, 2018.

[19] R. Ausavarungnirun, C. J. Rossbach, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, and O. Mutlu, "Improving Multi-Application Concurrency Support Within the GPU Memory System," arXiv:1708.04911 [cs.AR], 2017.

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

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

[22] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *ISCA*, 2013.

[23] A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," in *MICRO*, 2013.

[24] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," in *HPCA*, 2011.

[25] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB for Chip Multiprocessors," in *ASPLOS*, 2010.

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

[27] D. Bouvier and B. Sander, "Applying AMD's Kaveri APU for Heterogeneous Computing," in *Hot Chips*, 2014.

[28] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, 2012.

[29] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *HPCA*, 2017.

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

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

[32] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. W. Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *MICRO*, 2014.

[33] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W. W. Hwu, "Adaptive Cache Bypass and Insertion for Many-Core Accelerators," in *MES*, 2014.

[34] M. Clark, "A New Xc6 Core Architecture for the Next Generation of Computing," in *Hot Chips*, 2016.

[35] J. Cong, Z. Fang, Y. Hao, and G. Reinman, "Supporting Address Translation for Accelerator-Centric Architectures," in *HPCA*, 2017.

[36] G. Cox and A. Bhattacharjee, "Efficient Address Translation with Multiple Page Sizes," in *ASPLOS*, 2016.

[37] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *GPGPU*, 2010.

[38] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.

[39] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Aérgia: Exploiting Packet Latency Slack in On-Chip Networks," in *ISCA*, 2010.

[40] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters," in *HPCS*, 2010.

[41] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-core Systems," in *MICRO*, 2009.

[42] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.

[43] S. Eyerman and L. Eeckhout, "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance," *CAL*, 2014.

[44] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *ISCA*, 2011.

[45] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.

[46] A. Herrera, "NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation," NVIDIA White Paper, 2014.

[47] Intel Corp., "Intel® Microarchitecture Codename Sandy Bridge," http://www.intel.com/technology/architecture-silicon/2ndgen/.

[48] Intel Corp., "Product Speficiations: Products Formerly Ivy Bridge," http://ark.intel.com/products/codename/29902/, 2012.

[49] Intel Corp., "Introduction to Intel Architecture," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf, 2014.

[50] Intel Corp., "Intel 64 and IA-32 Architectures Software Developers Manual," 2016, https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[51] Intel Corp., "Intel Virtualization Technology for Directed I/O," http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, 2016.

[52] Intel Corp., "Intel® 64 and IA-32 Architectures Optimization Reference Manual," 2016.

[53] Intel Corp., "6th Generation Intel Core Processor Family Datasheet, Vol. 1," http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/desktop-6th-gen-core-family-datasheet-vol-1.pdf, 2017.

[54] B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors," in *IEEE Micro*, 1998.

[55] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive Insertion Policies for Managing Shared Caches," in *PACT*, 2008.

[56] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *ISCA*, 2010.

[57] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.

[58] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *HPCA*, 2014.

[59] A. Jog, "Design and Analysis of Scheduling Techniques for Throughput Processors," Ph.D. dissertation, Pennsylvania State Univ., 2015.

[60] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.

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

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

[63] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced GPU Performance," in *SIGMETRICS*, 2016.

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

[65] I. Karlin, A. Bhatele, J. Keasler, B. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application," in *IPDPS*, 2013.
[66] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," 2013.
[67] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-Class GPU Resource Management in the Operating System," in *USENIX ATC*, 2012.
[68] O. Kayiran, N. Chidambaram, A. Jog, R. Ausavarungnirun, M. Kandemir, G. Loh, O. Mutlu, and C. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
[69] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *PACT*, 2013.
[70] O. Kayıran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
[71] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[72] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[73] B. Langmead and S. L. Salzberg, "Fast Gapped-Read Alignment with Bowtie 2," *Nature Methods*, 2012.
[74] J. Lee and H. Kim, "TAP: A TLP-Aware Cache Management Policy for a CPU–GPU Heterogeneous Architecture," in *HPCA*, 2012.
[75] J. Lee, M. Samadi, and S. Mahlke, "VAST: The Illusion of a Large Memory Space for GPUs," in *PACT*, 2014.
[76] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-Driven Dynamic GPU Cache Bypassing," in *ICS*, 2015.
[77] D. Li, M. Rhu, D. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Redder, "Priority-Based Cache Allocation in Throughput Processors," in *HPCA*, 2015.
[78] T. Li, V. K. Narayana, and T. El-Ghazawi, "Symbiotic Scheduling of Concurrent GPU Kernels for Performance and Energy Optimizations," in *CF*, 2014.
[79] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 2008.
[80] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "Accelerating Molecular Dynamics Simulations using Graphics Processing Units with CUDA," *Computer Physics Communications*, vol. 179, no. 9, pp. 634–641, 2008.
[81] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," in *TACO*, 2013.
[82] T. Mashimo, Y. Fukunishi, N. Kamiya, Y. Takano, I. Fukuda, and H. Nakamura, "Molecular Dynamics Simulations Accelerated by GPU for Biological Macromolecules with a Non-Ewald Scheme for Electrostatic Interactions," *Journal of Chemical Theory and Computation*, 2013.
[83] J. Menon, M. de Kruijf, and K. Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," in *ISCA*, 2012.
[84] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
[85] D. Mrozek, M. Brozek, and B. Malysiak-Mrozek, "Parallel Implementation of 3D Protein Structure Similarity Searches Using a GPU and the CUDA," *Journal of Molecular Modeling*, 2014.
[86] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
[87] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO*, 2007.
[88] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[89] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[90] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *MICRO*, 2011.
[91] M. S. Nobile, P. Cazzaniga, A. Tangherloni, and D. Besozzi, "Graphics Processing Units in Bioinformatics, Computational Biology and Systems Biology," *Briefings in Bioinformatics*, 2016.
[92] NVIDIA Corp., "NVIDIA Tegra K1," http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper-v1.0.pdf.
[93] NVIDIA Corp., "NVIDIA Tegra X1," https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.
[94] NVIDIA Corp., "CUDA C/C++ SDK Code Samples," http://developer.nvidia.com/cuda-cc-sdk-code-samples, 2011.
[95] NVIDIA Corp., "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2011.
[96] NVIDIA Corp., "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.
[97] NVIDIA Corp., "Tesla K40 GPU Active Accelerator," https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf, 2013.
[98] NVIDIA Corp., "NVIDIA GeForce GTX 750 Ti," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014.
[99] NVIDIA Corp., "Multi-Process Service," https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2015.
[100] NVIDIA Corp., "NVIDIA GeForce GTX 1080," https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2016.
[101] NVIDIA Corp., "NVIDIA Tesla P100," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.
[102] NVIDIA Corp., "CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html, 2017.
[103] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *ASPLOS*, 2013.
[104] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A Case for Toggle-Aware Compression for GPU Systems," in *HPCA*, 2016.
[105] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ASPLOS*, 2014.
[106] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *HPCA*, 2014.
[107] PowerVR, "PowerVR Hardware Architecture Overview for Developers," http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf, 2016.
[108] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *ISCA*, 2007.
[109] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006.
[110] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
[111] T. G. Rogers, "Locality and Scheduling in the Massively Multithreaded Era," Ph.D. dissertation, Univ. of British Columbia, 2015.
[112] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.
[113] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule them All," in *HPCA*, 2010.
[114] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions to Manage GPUs as Compute Devices," in *SOSP*, 2011.
[115] SAFARI Research Group, "Mosaic – GitHub Repository," https://github.com/CMU-SAFARI/Mosaic/.
[116] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT*, 2012.
[117] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," in *TACO*, 2015.
[118] SK Hynix Inc., "Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0," http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf.
[119] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *SPIE*, 1981.
[120] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," in *ICPP*, 1978.
[121] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Univ. of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.
[122] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," in *TPDS*, 2016.
[123] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
[124] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
[125] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
[126] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why Not Virtualizing GPUs at the Hypervisor?" in *USENIX ATC*, 2014.
[127] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *ISCA*, 2014.
[128] J. E. Thornton, "Parallel Operation in the Control Data 6600," *AFIPS FJCC*, 1964.
[129] J. E. Thornton, *Design of a Computer: The Control Data 6600.* Scott Foresman & Co, 1970.

[130] K. Tian, Y. Dong, and D. Cowperthwaite, "A Full GPU Virtualization Solution with Mediated Pass-Through," in *USENIX ATC*, 2014.

[131] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "SQUASH: Simple QoS-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," arXiv:1505.07502 [cs.AR], 2015.

[132] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," in *TACO*, 2016.

[133] H. Vandierendonck and A. Seznec, "Fairness Metrics for Multi-Threaded Processors," *CAL*, 2011.

[134] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems," in *ISPASS*, 2016.

[135] T. Vijayaraghavany, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, "Design and Analysis of an APU for Exascale Computing," in *HPCA*, 2017.

[136] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *MICRO*, 2016.

[137] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.

[138] N. Vijaykumar, G. Pekhimenko, A. Jog, S. Ghose, A. Bhowmick, R. Ausavarungnirun, C. R. Das, M. T. Kandemir, T. C. Mowry, and O. Mutlu, "A Framework for Accelerating Bottlenecks in GPU Execution with Assist Warps," arXiv:1602.01348 [cs.AR], 2016.

[139] Vivante, "Vivante Vega GPGPU Technology," http://www.vivantecorp.com/index.php/en/technology/gpgpu.html, 2016.

[140] L. Vu, H. Sivaraman, and R. Bidarkar, "GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor," in *HPC*, 2014.

[141] Z. Wang, J. Yang, R. Melhem, B. R. Childers, Y. Zhang, and M. Guo, "Simultaneous Multikernel GPU: Multi-Tasking Throughput Processors via Fine-Grained Sharing," in *HPCA*, 2016.

[142] S. Wasson, "AMD's A8-3800 Fusion APU," http://techreport.com/articles.x/21730, 2011.

[143] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *ISPASS*, 2010.

[144] C.-J. Wu and M. Martonosi, "Characterization and Dynamic Mitigation of Intra-application Cache Interference," in *ISPASS*, 2011.

[145] X. Xie, Y. Liang, G. Sun, and D. Chen, "An Efficient Compiler Framework for Cache Bypassing on GPUs," in *ICCAD*, 2013.

[146] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *HPCA*, 2015.

[147] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming," in *ISCA*, 2016.

[148] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks," in *PPoPP*, 2017.

[149] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," in *MICRO*, 2017.

[150] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures," in *MICRO*, 2009.

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

[152] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," U.S. Patent Number 5,630,096, 1997.