Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes

Rachata Ausavarungnirun[†] Joshua Landgraf[‡] Vance Miller[‡] Saugata Ghose[†]
Jayneel Gandhi[ℵ] Christopher J. Rossbach[‡ℵ] Onur Mutlu[§†]

[†]Carnegie Mellon University [‡]University of Texas at Austin [ℵ]VMware Research [§]ETH Zürich

ABSTRACT

Contemporary discrete GPUs support rich memory management features such as virtual memory and demand paging. These features simplify GPU programming by providing a virtual address space abstraction similar to CPUs and eliminating manual memory management, but they introduce high performance overheads during (1) address translation and (2) page faults. A GPU relies on high degrees of *thread-level parallelism* (TLP) to hide memory latency. Address translation can undermine TLP, as a single miss in the translation lookaside buffer (TLB) invokes an expensive serialized page table walk that often stalls *multiple* threads. Demand paging can also undermine TLP, as multiple threads often stall while they wait for an expensive data transfer over the system I/O (e.g., PCIe) bus when the GPU demands a page.

In modern GPUs, we face a trade-off on how the page size used for memory management affects address translation and demand paging. The address translation overhead is lower when we employ a *larger* page size (e.g., 2MB *large pages*, compared with conventional 4KB *base pages*), which increases TLB coverage and thus reduces TLB misses. Conversely, the demand paging overhead is lower when we employ a *smaller* page size, which decreases the system I/O bus transfer latency. Support for *multiple page sizes* can help relax the page size trade-off so that address translation and demand paging optimizations work together synergistically. However, existing page *coalescing* (i.e., merging base pages into a large page) and *splintering* (i.e., splitting a large page into base pages) policies require costly base page migrations that undermine the benefits multiple page sizes provide. In this paper, we observe that GPGPU applications present an opportunity to support multiple page sizes without costly data migration, as the applications perform most of their memory allocation *en masse* (i.e., they allocate a large number of base pages at once). We show that this *en masse* allocation allows us to create intelligent memory allocation policies which ensure that base pages that are contiguous in virtual memory are allocated to contiguous physical memory pages. As a result, coalescing and splintering operations no longer need to migrate base pages.

We introduce *Mosaic*, a GPU memory manager that provides *application-transparent* support for multiple page sizes. *Mosaic* uses base pages to transfer data over the system I/O bus, and allocates physical memory in a way that (1) preserves base page contiguity and (2) ensures that a large page frame contains pages from only a single memory protection domain. We take advantage of this allocation strategy to design a novel in-place page size selection mechanism that avoids data migration. This mechanism allows the TLB to use large pages, reducing address translation overhead. During data transfer, this mechanism enables the GPU to transfer *only* the base pages that are needed by the application over the system I/O bus, keeping demand paging overhead low. Our evaluations show that *Mosaic* reduces address translation overheads while efficiently achieving the benefits of demand paging, compared to a contemporary GPU that uses only a 4KB page size. Relative to a state-of-the-art GPU memory manager, *Mosaic* improves the performance of homogeneous and heterogeneous multi-application workloads by 55.5% and 29.7% on average, respectively, coming within 6.8% and 15.4% of the performance of an ideal TLB where all TLB requests are hits.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**;

KEYWORDS

graphics processing units, GPGPU applications, address translation, demand paging, large pages, virtual memory management

ACM Reference format:
R. Ausavarungnirun et al. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of The 50th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, MA, USA, October 14–18, 2017 (MICRO-50),* 15 pages.

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9…$15.00
https://doi.org/10.1145/3123939.3123975

1 INTRODUCTION

Graphics Processing Units (GPUs) are used for an ever-growing range of application domains due to steady increases in GPU compute density and continued improvements in programming tools [55, 60, 80]. The growing adoption of GPUs has in part been due to better high-level language support [20, 80, 97, 110], which has improved GPU programmability. Recent support within GPUs for *memory virtualization* features, such as a unified virtual address space [60, 77], demand paging [82], and preemption [2, 82], can provide *fundamental* improvements that can ease programming. These features allow developers to exploit key benefits that have long been taken for granted in CPUs (e.g., application portability, multi-application execution). Such familiar features can dramatically improve programmer productivity and further boost GPU adoption. However, a number of challenges have kept GPU memory virtualization from achieving performance similar to that in CPUs [66, 114]. In this work, we focus on two fundamental challenges: (1) the address translation challenge, and (2) the demand paging challenge.

Address Translation Challenge. Memory virtualization relies on *page tables* to store virtual-to-physical address translations. Conventionally, systems store one translation for every *base page* (e.g., a 4KB page). To translate a virtual address on demand, a series of *serialized* memory accesses are required to traverse (i.e., *walk*) the page table [91, 92]. These serialized accesses clash with the single-instruction multiple-thread (SIMT) execution model [33, 63, 73]

used by GPU-based systems, which relies on *high degrees of concurrency* through *thread-level parallelism* (TLP) to hide long memory latencies during GPU execution. *Translation lookaside buffers* (TLBs) can reduce the latency of address translation by caching recently-used address translation information. Unfortunately, as application working sets and DRAM capacity have increased in recent years, state-of-the-art GPU TLB designs [91, 92] suffer due to *inter-application* interference and stagnant TLB sizes. Consequently, GPUs have *poor TLB reach*, i.e., the TLB covers only a small fraction of the physical memory working set of an application. Poor TLB reach is particularly detrimental with the SIMT execution model, as a *single* TLB miss can stall *hundreds* of threads at once, undermining TLP within a GPU and significantly reducing performance [66, 114].

Large pages (e.g., the 2MB or 1GB pages in modern CPUs [41, 43]) can significantly reduce the overhead of address translation. A major constraint for TLB reach is the small, fixed number of translations that a TLB can hold. If we store one translation for every large page instead of one translation for every base page, the TLB can cover a much larger fraction of the virtual address space using the same number of page translation entries. Large pages have been supported by CPUs for decades [103, 104], and large page support is emerging for GPUs [91, 92, 118]. However, large pages increase the risk of *internal fragmentation*, where a portion of the large page is unallocated (or unused). Internal fragmentation occurs because it is often difficult for an application to completely utilize large contiguous regions of memory. This fragmentation leads to (1) *memory bloat*, where a much greater amount of physical memory is allocated than the amount of memory that the application needs; and (2) longer memory access latencies, due to a lower effective TLB reach and more page faults [59].

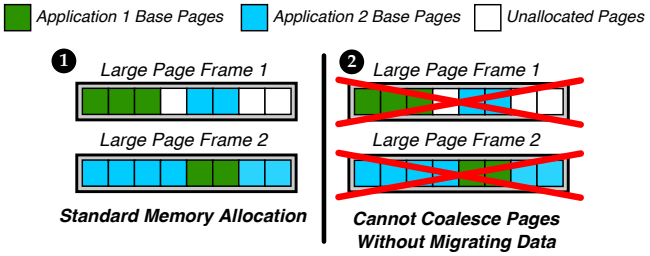
Demand Paging Challenge. For *discrete GPUs* (i.e., GPUs that are not in the same package/die as the CPU), *demand paging* can incur significant overhead. With demand paging, an application can request data that is not currently resident in GPU memory. This triggers a *page fault*, which requires a long-latency data transfer for an entire page over the system I/O bus, which, in today's systems, is also called the PCIe bus [85]. A single page fault can cause multiple threads to stall at once, as threads often access data in the same page due to data locality. As a result, the page fault can significantly reduce the amount of TLP that the GPU can exploit, and the long latency of a page fault harms performance [118].

Unlike address translation, which benefits from *larger* pages, demand paging benefits from *smaller pages*. Demand paging for large pages requires a greater amount of data to be transferred over the system I/O bus during a page fault than for conventional base pages. The larger data transfer size increases the transfer time significantly, due to the long latency and limited bandwidth of the system I/O bus. This, in turn, significantly increases the amount of time that GPU threads stall, and can further decrease the amount of TLP. To make matters worse, as the size of a page increases, there is a greater probability that an application does not need all of the data in the page. As a result, threads may stall for a longer time without gaining any further benefit in return.

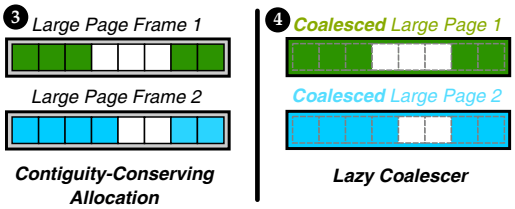
Page Size Trade-Off. We find that memory virtualization in state-of-the-art GPU systems has a fundamental trade-off due to the page size choice. A *larger* page size reduces address translation stalls by increasing TLB reach and reducing the number of high-latency TLB misses. In contrast, a *smaller* page size reduces demand paging stalls by decreasing the amount of unnecessary data transferred over the system I/O bus [92, 118]. We can relax the page size trade-off by using *multiple* page sizes *transparently* to the application, and, thus, to the programmer. In a system that supports multiple page sizes, several base pages that are *contiguous in both virtual and physical memory* can be *coalesced* (i.e., combined) into a single large page, and a large page can be *splintered* (i.e., split) into multiple base pages. With multiple page sizes, and the ability to change virtual-to-physical mappings dynamically, the GPU system can support good TLB reach by using large pages for address translation, while providing better demand paging performance by using base pages for data transfer.

Application-transparent support for multiple page sizes has proven challenging for CPUs [59, 74]. A key property of memory virtualization is to enforce *memory protection*, where a distinct virtual address space (i.e., a *memory protection domain*) is allocated to an individual application or a virtual machine, and memory is shared *safely* (i.e., only with explicit permissions for accesses across different address spaces). In order to ensure that memory protection guarantees are not violated, coalescing operations can combine contiguous physical base pages into a single physical large page *only if all base pages belong to the same virtual address space.*

Unfortunately, in both CPU and state-of-the-art GPU memory managers, existing memory access patterns and allocation mechanisms make it difficult to find regions of physical memory where base pages can be coalesced. We show an example of this in Figure 1a, which illustrates how a state-of-the-art GPU memory manager [92] allocates memory for two applications. Within a single *large page frame* (i.e., a contiguous piece of physical memory that is the size of a large page and whose starting address is page aligned), the GPU memory manager allocates base pages from both Applications 1 and 2 (❶ in the figure). As a result, the memory manager *cannot* coalesce the base pages into a large page (❷) without first migrating some of the base pages, which would incur a high latency.



(a) State-of-the-art GPU memory management [92].



(b) Memory management with *Mosaic*.

Figure 1: Page allocation and coalescing behavior of GPU memory managers: (a) state-of-the-art [92], (b) *Mosaic*.

We make a *key observation* about the memory behavior of contemporary general-purpose GPU (GPGPU) applications. The vast majority of memory allocations in GPGPU applications are performed *en masse* (i.e., a large number of pages are allocated at the same time). The *en masse* memory allocation presents us with an opportunity: with so many pages being allocated at once, we can rearrange how we allocate the base pages to ensure that (1) *all* of the base pages allocated within a large page frame belong to the *same* virtual address space, and (2) base pages that are contiguous in virtual memory are allocated to a contiguous portion

of physical memory and aligned within the large page frame. *Our goal* in this work is to develop an application-transparent memory manager that performs such memory allocation, and uses this allocation property to efficiently support multiple page sizes in order to improve TLB reach and efficiently support demand paging.

To this end, we present *Mosaic*, a new GPU memory manager that uses our key observation to provide application-transparent support for multiple page sizes in GPUs while avoiding high overhead for coalescing and splintering pages. The key idea of *Mosaic* is to (1) transfer data to GPU memory at the small base page (e.g., 4KB) granularity, (2) allocate physical base pages in a way that avoids the need to migrate data during coalescing, and (3) use a simple coalescing mechanism to combine base pages into large pages (e.g., 2MB) and thus increase TLB reach. Figure 1b shows a high-level overview of how *Mosaic* allocates and coalesces pages. *Mosaic* consists of three key design components: (1) ***Co**ntiguity-**Co**nserving **A**llocation* (CoCoA), a memory allocator which provides a *soft guarantee* that *all* of the base pages within the same large page range belong to only a single application (❸ in the figure); (2) *In-Place Coalescer*, a page size selection mechanism that merges base pages into a large page immediately after allocation (❹), and thus does *not* need to monitor base pages to make coalescing decisions or migrate base pages; and (3) ***C**ontiguity-**A**ware **C**ompaction* (CAC), a memory compaction mechanism that transparently migrates data to avoid internal fragmentation within a large page frame, which frees up large page frames for CoCoA.

Key Results. We evaluate *Mosaic* using 235 workloads. Each workload consists of multiple GPGPU applications from a wide range of benchmark suites. Our evaluations show that compared to a contemporary GPU that uses only 4KB base pages, a GPU with *Mosaic* reduces address translation overheads while efficiently achieving the benefits of demand paging, thanks to its use of multiple page sizes. When we compare to a GPU with a state-of-the-art memory manager (see Section 3.1), we find that a GPU with *Mosaic* provides an average speedup of 55.5% and 29.7% for homogeneous and heterogeneous multi-application workloads, respectively, and comes within 6.8% and 15.4% of the performance of a GPU with an ideal TLB, where all TLB requests are hits. Thus, by alleviating the page size trade-off between address translation and demand paging overhead, *Mosaic* improves the efficiency and practicality of multi-application execution on the GPU.

This paper makes the following contributions:

- We analyze fundamental trade-offs on choosing the correct page size to optimize both address translation (which benefits from larger pages) and demand paging (which benefits from smaller pages). Based on our analyses, we motivate the need for application-transparent support of *multiple* page sizes in a GPU.
- We present *Mosaic*, a new GPU memory manager that *efficiently* supports multiple page sizes. *Mosaic* uses a novel mechanism to allocate contiguous virtual pages to contiguous physical pages in the GPU memory, and exploits this property to coalesce contiguously-allocated base pages into a large page for address translation with low overhead and no data migration, while still using base pages during demand paging.
- We show that *Mosaic*'s application-transparent support for multiple page sizes effectively improves TLB reach while efficiently achieving the benefits of demand paging. Overall, *Mosaic* improves the average performance of homogeneous and heterogeneous multi-application workloads by 55.5% and 29.7%, respectively, over a state-of-the-art GPU memory manager.

2 BACKGROUND

We first provide necessary background on contemporary GPU architectures. In Section 2.1, we discuss the GPU execution model. In Section 2.2, we discuss state-of-the-art support for GPU memory virtualization.

2.1 GPU Execution Model

GPU applications use *fine-grained multithreading* [105, 106, 112, 113]. A GPU application is made up of thousands of threads. These threads are clustered into *thread blocks* (also known as *work groups*), where each thread block consists of multiple smaller bundles of threads that execute concurrently. Each such thread bundle is known as a *warp*, or a *wavefront*. Each thread within the warp executes the same instruction at the same program counter value. The GPU avoids stalls due to dependencies and long memory latencies by taking advantage of *thread-level parallelism* (TLP), where the GPU swaps out warps that have dependencies or are waiting on memory with other warps that are ready to execute.

A GPU consists of multiple *streaming multiprocessors* (SMs), also known as *shader cores*. Each SM executes one warp at a time using the single-instruction, multiple-thread (SIMT) execution model [33, 63, 73]. Under SIMT, all of the threads within a warp are executed in *lockstep*. Due to lockstep execution, a warp stalls when *any one thread* within the warp has to stall. This means that a warp is unable to proceed to the next instruction until the *slowest* thread in the warp completes the current instruction.

The GPU memory hierarchy typically consists of multiple levels of memory. In contemporary GPU architectures, each SM has a private data cache, and has access to one or more shared memory partitions through an interconnect (typically a crossbar). A *memory partition* combines a single slice of the banked L2 cache with a memory controller that connects the GPU to off-chip main memory (DRAM). More detailed information about the GPU memory hierarchy can be found in [8, 10, 44, 46, 47, 48, 54, 86, 95, 115, 116].

2.2 Virtualization Support in GPUs

Hardware-supported memory virtualization relies on address translation to map each virtual memory address to a physical address within the GPU memory. Address translation uses page-granularity virtual-to-physical mappings that are stored within a multi-level *page table*. To look up a mapping within the page table, the GPU performs a page table *walk*, where a page table walker traverses through each level of the page table in main memory until the walker locates the *page table entry* for the requested mapping in the last level of the table. GPUs with virtual memory support have *translation lookaside buffers* (TLBs), which cache page table entries and avoid the need to perform a page table walk for the cached entries, thus reducing the address translation latency.

The introduction of address translation hardware into the GPU memory hierarchy puts TLB misses on the critical path of application execution, as a TLB miss invokes a page table walk that can stall multiple threads and degrade performance significantly. (We study the impact of TLB misses and page table walks in Section 3.1.) A GPU uses multiple TLB levels to reduce the number of TLB misses, typically including private per-SM L1 TLBs and a shared L2 TLB [91, 92, 118]. Traditional address translation mechanisms perform memory mapping using a *base page* size of 4KB. Prior work for *integrated GPUs* (i.e., GPUs that are in the same package or die as the CPU) has found that using a larger page size can improve address translation performance by improving *TLB reach* (i.e., the maximum fraction of memory that can be accessed using the cached TLB entries) [91, 92, 118]. For a TLB that holds a fixed number of page table entries, using the *large page* (e.g., a

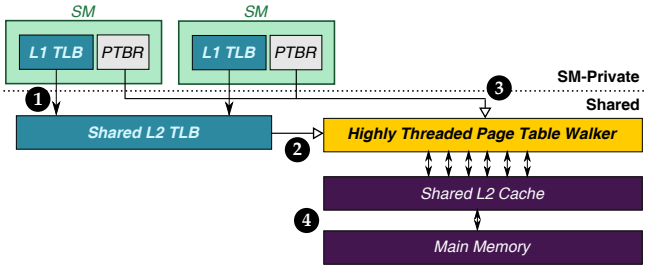


Figure 2: GPU-MMU baseline design with a two-level TLB.

page with a size of 2MB or greater) as the granularity for mapping greatly increases the TLB reach, and thus reduces the TLB miss rate, compared to using the base page granularity. While memory hierarchy designs for widely-used GPU architectures from NVIDIA, AMD, and Intel are not publicly available, it is widely accepted that contemporary GPUs support TLB-based address translation and, in some models, large page sizes [2, 66, 77, 78, 79]. To simplify translation hardware in a GPU that uses multiple page sizes (i.e., both base pages and large pages), we assume that each TLB level contains two separate sets of entries [35, 50, 51, 84, 89, 90], where one set of entries stores only base page translations, while the other set of entries stores only large page translations.

State-of-the-art GPU memory virtualization provides support for *demand paging* [3, 60, 82, 92, 118]. In demand paging, all of the memory used by a GPU application does not need to be transferred to the GPU memory at the beginning of application execution. Instead, during application execution, when a GPU thread issues a memory request to a page that has not yet been allocated in the GPU memory, the GPU issues a *page fault*, at which point the data for that page is transferred over the off-chip system I/O bus (e.g., the PCIe bus [85] in contemporary systems) from the CPU memory to the GPU memory. The transfer requires a long latency due to its use of an off-chip bus. Once the transfer completes, the GPU runtime allocates a physical GPU memory address to the page, and the thread can complete its memory request.

3 A CASE FOR MULTIPLE PAGE SIZES

Despite increases in DRAM capacity, TLB capacity (i.e., the number of cached page table entries) has not kept pace, and thus TLB reach has been declining. As a result, address translation overheads have started to significantly increase the execution time of many large-memory workloads [15, 34, 66, 91, 92, 114]. In this section, we (1) analyze how the address translation overhead changes if we use large pages instead of base pages, and (2) examine the advantages and disadvantages of both page sizes.

3.1 Effect of Page Size on TLB Performance

To quantify the performance trade-offs between base and large pages, we simulate a number of recently-proposed TLB designs that support demand paging [92, 118] (see Section 5 for our methodology). We slightly modify Power et al.'s TLB design [92] to create our baseline, which we call *GPU-MMU*. Power et al. [92] propose a GPU memory manager that has a private 128-entry L1 TLB for each SM , a highly-threaded page table walker, and a page walk cache [92]. From our experiments, we find that using a shared L2 TLB instead of a page walk cache increases the average performance across our workloads (described in Section 5) by 14% (not shown). As a result, our GPU-MMU baseline design (shown in Figure 2) omits the page walk cache in favor of a 512-entry shared L2 TLB.

In our GPU-MMU baseline design, a shared L2 TLB entry is extended with address space identifiers. TLB accesses from multiple threads to the same page are coalesced (i.e., combined). On an L1 TLB miss (❶ in Figure 2), the shared L2 TLB is accessed. If the request misses in the shared L2 TLB, the page table walker begins a walk (❷). The walker reads the Page Table Base Register (PTBR)[1] from the core that caused the TLB miss (❸), which contains a pointer to the root of the page table. The walker then accesses each level of the page table, retrieving the page table data from either the shared L2 cache or the GPU main memory (❹).

Figure 3 shows the performance of two GPU-MMU designs: (1) a design that uses the base 4KB page size, and (2) a design that uses a 2MB large page size, where both designs have *no demand paging overhead* (i.e., the system I/O bus transfer takes zero cycles to transfer a page). We normalize the performance of the two designs to a GPU with an ideal TLB, where *all* TLB requests hit in the L1 TLB. We make two observations from the figure.

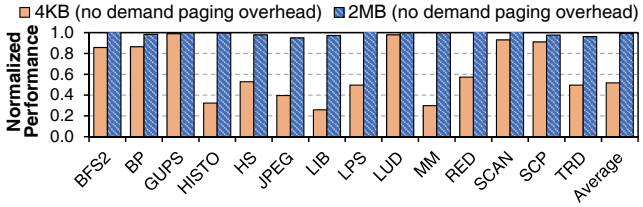


Figure 3: Performance of a GPU with *no demand paging overhead*, using (1) 4KB base pages and (2) 2MB large pages, normalized to the performance of a GPU with an ideal TLB.

First, compared to the ideal TLB, the GPU-MMU with 4KB base pages experiences an average performance loss of 48.1%. We observe that with 4KB base pages, a single TLB miss often stalls *many* of the warps, which undermines the latency hiding behavior of the SIMT execution model used by GPUs. Second, the figure shows that using a 2MB page size with the same number of TLB entries as the 4KB design allows applications to come within 2% of the ideal TLB performance. We find that with 2MB pages, the TLB has a much larger reach, which reduces the TLB miss rate substantially. Thus, there is strong incentive to use large pages for address translation.

3.2 Large Pages Alone Are Not the Answer

A natural solution to consider is to use *only* large pages for GPU memory management. Using only large pages would reduce address translation overhead significantly, with minimal changes to the hardware or runtime. Unfortunately, this solution is impractical because large pages (1) greatly increase the data transfer size of *each* demand paging request, causing contention on the system I/O bus, and harming performance; and (2) waste memory by causing memory bloat due to internal fragmentation.

Demand Paging at a Large Page Granularity. Following the nomenclature from [118], we denote GPU-side page faults that induce demand paging transfers across the system I/O bus as *far-faults*. Prior work observes that while a 2MB large page size reduces the number of far-faults in GPU applications that exhibit locality, the *load-to-use latency* (i.e., the time between when a thread issues a load request and when the data is returned to the thread) increases significantly when a far-fault does occur [118]. The impact of far-faults is particularly harmful for workloads with high locality, as *all* warps touching the 2MB *large page frame* (i.e., a contiguous, page-aligned 2MB region of physical memory) must stall, which limits the GPU's ability to overlap the system I/O bus transfer by executing other warps. Based on PCIe latency measurements from a real GTX 1080 system [83], we determine that the load-to-use

[1] CR3 in the x86 ISA [42], TTB in the ARM ISA [7].

latency with 2MB large pages (318 µs) is six times the latency with 4KB base pages (55 µs).

Figure 4 shows how the GPU performance changes when we use different page sizes and include the effect of the demand paging overhead (see Section 5 for our methodology). We make three observations from the figure. First, for 4KB base pages, the demand paging overhead reduces performance, by an average of 40.0% for our single-application workloads, and 82.3% for workloads with five concurrently-executing applications. Second, for our single-application workloads, we find that with demand paging overhead, 2MB pages slow down the execution time by an average of 92.5% compared to 4KB pages with demand paging, as the GPU cores now spend most of their time stalling on the system I/O bus transfers. Third, the overhead of demand paging for larger pages gets significantly worse as more applications share the GPU. With two applications concurrently executing on the GPU, the average performance degradation of demand paging with 2MB pages instead of 4KB pages is 98.0%, and with five applications, the average degradation is 99.8%.

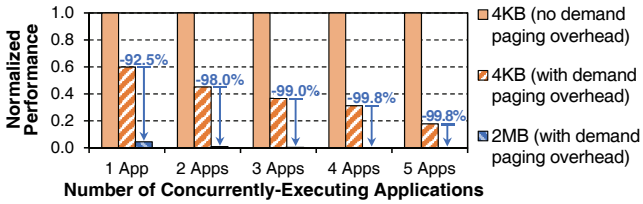


Figure 4: Performance impact of system I/O bus transfer during demand paging for base pages and large pages, normalized to base page performance with no demand paging overhead.

Memory Bloat. Large pages expose the system to internal fragmentation and *memory bloat*, where a much greater amount of physical memory is allocated than the amount of memory actually needed by an application. To understand the impact of memory bloat, we evaluate the amount of memory allocated to each application when run in isolation, using 4KB and 2MB page sizes. When we use the 4KB base page size, our applications have working sets ranging from 10MB to 362MB, with an average of 81.5MB (see Section 5 and [11] for details). We find that the amount of allocated memory inflates by 40.2% on average, and up to 367% in the worst case, when we use 2MB pages instead of 4KB pages (not shown). These numbers are likely conservative, as we expect that the fragmentation would worsen as an application continues to run for longer time scales than we can realistically simulate. Such waste is unacceptable, particularly when there is an increasing demand for GPU memory due to other concurrently-running applications.

We conclude that despite the potential performance gain of 2MB large pages (when the overhead of demand paging is ignored), the demand paging overhead actually causes 2MB large pages to perform *much worse* than 4KB base pages. As a result, it is impractical to use *only* 2MB large pages in the GPU. Therefore, a design that delivers the best of both page sizes is needed.

3.3 Challenges for Multiple Page Size Support

As Sections 3.1 and 3.2 demonstrate, we cannot efficiently optimize GPU performance by employing only a single page size. Recent works on TLB design for integrated GPUs [91, 92] and on GPU demand paging support [3, 60, 82, 92, 118] corroborate our own findings on the performance cost of address translation and the performance opportunity of large pages. **Our goal** is to design a new memory manager for GPUs that efficiently supports *multiple page sizes*, to exploit the benefits of both small and large page sizes, while avoiding the disadvantages of each. In order to (1) not burden programmers and (2) provide performance improvements for legacy applications, we would like to enable multiple page size support *transparently to the application*. This constraint introduces several design challenges that must be taken into account.

Page Size Selection. While conceptually simple, multiple page size support introduces complexity for memory management that has traditionally been difficult to handle. Despite architectural support within CPUs [59, 74] for several decades, the adoption of multiple page sizes has been quite slow and application-domain specific [15, 34]. The availability of large pages can either be exposed to application programmers, or managed transparently to an application. Application-exposed management forces programmers to reason about physical memory and use specialized APIs [6, 68] for page management, which usually sacrifices code portability and increases programmer burden. In contrast, application-transparent support (e.g., management by the OS) requires no changes to existing programs to use large pages, but it does require the memory manager to make *predictive* decisions about whether applications would benefit from large pages. OS-level large page management remains an active research area [59, 74], and the optimization guidance for many modern applications continues to advise strongly *against* using large pages [1, 25, 65, 69, 87, 93, 107, 117], due to high-latency data transfers over the system I/O bus and memory bloat (as described in Section 3.2). In order to provide effective application-transparent support for multiple page sizes in GPUs, we must develop a policy for selecting page sizes that avoids high-latency data transfer over the system I/O bus, and does not introduce significant memory bloat.

Hardware Implementation. Application-transparent support for multiple page sizes requires (1) primitives that implement the transition between different page sizes, and (2) mechanisms that create and preserve contiguity in both the virtual and physical address spaces. We must add support in the GPU to *coalesce* (i.e., combine) multiple base pages into a single large page, and *splinter* (i.e., split) a large page back into multiple base pages. While the GPU memory manager can *migrate* base pages in order to create opportunities for coalescing, base page migration incurs a high latency overhead [21, 101]. In order to avoid the migration overhead without sacrificing coalescing opportunities, the GPU needs to initially *allocate* data in a *coalescing-friendly manner*.

GPUs face additional implementation challenges over CPUs, as they rely on hardware-based memory allocation mechanisms and management. In CPU-based application-transparent large page management, coalescing and splintering are performed by the operating system [59, 74], which can (1) use locks and inter-processor interrupts (IPIs) to implement atomic updates to page tables, (2) stall any accesses to the virtual addresses whose mappings are changing, and (3) use background threads to perform coalescing and splintering. GPUs currently have no mechanism to atomically move pages or change page mappings for coalescing or splintering.

4 MOSAIC

In this section, we describe *Mosaic*, a GPU memory manager that provides application-transparent support for multiple page sizes and solves the challenges that we discuss in Section 3.3. At runtime, *Mosaic* (1) *allocates* memory in the GPU such that base pages that are contiguous in virtual memory are contiguous within a large page frame in physical memory (which we call *contiguity-conserving allocation*; Section 4.2); (2) *coalesces* base pages into a large page frame as soon as the data is allocated, only if all of the pages are *i)* contiguous in both virtual and physical memory, and *ii)* belong to

the same application (Section 4.3); and (3) *compacts* a large page (i.e., moves the allocated base pages within the large page frame to make them contiguous) if internal fragmentation within the page is high after one of its constituent base pages is deallocated (Section 4.4).

4.1 High-Level Overview of Mosaic

Figure 5 shows the major components of *Mosaic*, and how they interact with the GPU memory. *Mosaic* consists of three components: ***Co**ntiguity-**Co**nserving **A**llocation* (CoCoA), the *In-Place Coalescer*, and ***C**ontiguity-**A**ware **C**ompaction* (CAC). These three components work together to coalesce (i.e., combine) and splinter (i.e., split apart) base pages to/from large pages during memory management. Memory management operations for *Mosaic* take place at two times: (1) when memory is *allocated*, and (2) when memory is *deallocated*.

Memory Allocation. When a GPGPU application wants to access data that is not currently in the GPU memory, it sends a request to the GPU runtime (e.g., OpenCL, CUDA runtimes) to transfer the data from the CPU memory to the GPU memory (❶ in Figure 5). A GPGPU application typically allocates a large number of base pages at the same time. CoCoA allocates space within the GPU memory (❷) for the base pages, working to conserve the contiguity of base pages, if possible during allocation. Regardless of contiguity, CoCoA provides a *soft guarantee* that a single large page frame contains base pages from only a single application. Once the base page is allocated, CoCoA initiates the data transfer across the system I/O bus (❸). When the data transfer is complete (❹), CoCoA notifies the *In-Place Coalescer* that allocation is done by sending a list of the large page frame addresses that were allocated (❺). For each of these large page frames, the runtime portion of the *In-Place Coalescer* then checks to see whether (1) *all* base pages within the large page frame have been allocated, and (2) the base pages within the large page frame are contiguous in both virtual and physical memory. If both conditions are true, the hardware portion of the *In-Place Coalescer* updates the page table to coalesce the base pages into a large page (❻).

Memory Deallocation. When a GPGPU application wants to deallocate memory (e.g., when an application kernel finishes), it sends a deallocation request to the GPU runtime (❼). For all deallocated base pages that are coalesced into a large page, the runtime invokes CAC for the corresponding large page. The runtime portion of CAC checks to see whether the large page has a high degree of *internal fragmentation* (i.e., if the number of unallocated base pages within the large page exceeds a predetermined threshold). For each large page with high internal fragmentation, the hardware portion of CAC updates the page table to splinter the large page back into its constituent base pages (❽). Next, CAC compacts the splintered large page frames, by migrating data from multiple splintered large page frames into a single large page frame (❾). Finally, CAC notifies CoCoA of the large page frames that are now free after compaction (❿), which CoCoA can use for future memory allocations.

4.2 Contiguity-Conserving Allocation

Base pages can be coalesced into a large page frame only if (1) all base pages within the frame are contiguous in both virtual and physical memory, (2) the data within the large page frame is page aligned with the corresponding large page within virtual memory (i.e., the first base page within the large page frame is also the first base page of a virtual large page), and (3) all base pages within the frame come from the same virtual address space (e.g., the same application, or the same virtual machine). As Figure 1a (see Section 1) shows, traditional memory managers allocate base pages without conserving contiguity or ensuring that the base pages within a large page frame belong to the same application. For example, if the memory manager wants to coalesce base pages of Application 1 into a large page frame (e.g., Large Page Frame 1), it must first migrate Application 2's base pages to *another* large page frame, and may need to migrate some of Application 1's base pages within the large page frame to create contiguity. Only after this data migration, the base pages would be ready to be coalesced into a large page frame.

In *Mosaic*, we minimize the overhead of coalescing pages by designing CoCoA to take advantage of the memory allocation behavior of GPGPU applications. Similar to many data-intensive applications [37, 75], GPGPU applications typically allocate memory *en masse* (i.e., they allocate a large number of pages at a time). The *en masse* allocation takes place when an application kernel is about to be launched, and the allocation requests are often for a large contiguous region of virtual memory. This region is much larger than the large page size (e.g., 2MB), and *Mosaic* allocates multiple page-aligned 2MB portions of contiguous virtual memory from the region to large page frames in physical memory, as shown in Figure 1b (see Section 1). With CoCoA, the large page frames for Application 1 and Application 2 are ready to be coalesced as soon as their base pages are allocated, *without* the need for any data migration. For all other base pages (e.g., base pages not aligned in the virtual address space, allocation requests that are smaller than a large page), *Mosaic* simply allocates these pages to any free page, and does not exploit any contiguity.

Mosaic provides a *soft guarantee* that *all* base pages within a large page frame belong to the same application, which reduces the cost of performing coalescing and compaction, and ensures that these operations do not violate memory protection. To meet this guarantee during allocation, CoCoA needs to track the application

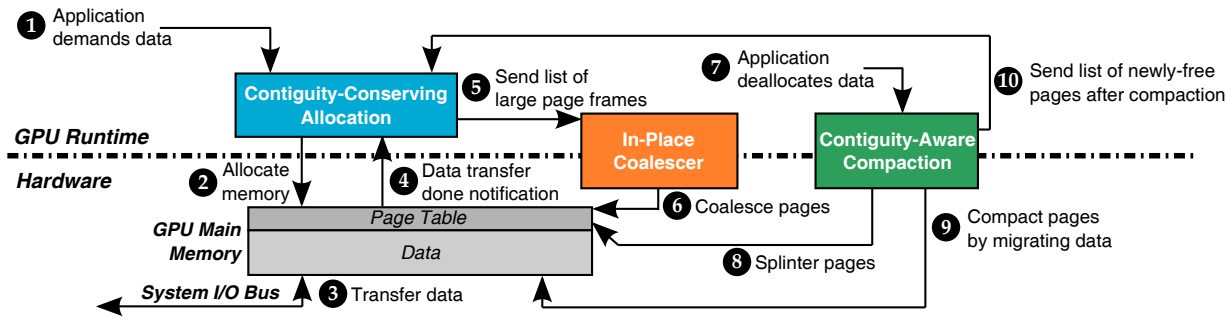


Figure 5: High-level overview of *Mosaic*, showing how and when its three components interact with the GPU memory.

that each large page frame with unallocated base pages is assigned to. The allocator maintains two sets of lists to track this information: (1) the *free frame list*, a list of free large page frames (i.e., frames where no base pages have been allocated) that are not yet assigned to any application; and (2) *free base page lists*, per-application lists of free base pages within large page frames where some (but not all) base pages are allocated. When CoCoA allocates a page-aligned 2MB region of virtual memory, it takes a large page frame from the free frame list and maps the virtual memory region to the frame. When CoCoA allocates base pages in a manner such that it cannot exploit contiguity, it takes a page from the free base page list for the application performing the memory request, to ensure that the soft guarantee is met. If the free base page list for an application is empty, CoCoA removes a large page frame from the free frame list, and adds the frame's base pages to the free base page list.

Note that there may be cases where the free frame list runs out of large page frames for allocation. We discuss how *Mosaic* handles such situations in Section 4.4.

4.3 In-Place Coalescer

In *Mosaic*, due to CoCoA (Section 4.2), we find that we can simplify how the page size is selected for each large page frame (i.e., decide which pages should be coalesced), compared to state-of-the-art memory managers. In state-of-the-art memory managers, such as our GPU-MMU baseline based on Power et al. [92], there is no guarantee that base pages within a large page frame belong to the *same* application, and memory allocators do not conserve virtual memory contiguity in physical memory. As a result, state-of-the-art memory managers must perform four steps to coalesce pages, as shown under the *Baseline* timeline in Figure 6a. First, the manager must identify opportunities for coalescing *across* multiple pages (not shown in the timeline, as this can be performed in the background). This is done by a hardware memory management unit (MMU), such as the Falcon coprocessor in recent GPU architectures [81], which tallies page utilization information from the page table entries of each base page. The most-utilized contiguous base pages are chosen for coalescing (*Pages A–G* in Figure 6a). Second, the manager must identify a large page frame where the coalesced base pages will reside, and then migrate the base pages to this new large page frame, which uses DRAM channel bandwidth (❶ in the figure). Third, the manager must update the page table entries (PTEs) to reflect the coalescing, which again uses DRAM channel bandwidth (❷). Fourth, the manager invokes a TLB flush to invalidate stale virtual-to-physical mappings (which point to the base page locations prior to migration), during which the SMs stall (❸). Thus, coalescing using a state-of-the-art memory manager causes significant DRAM channel utilization and SM stalls, as Figure 6a shows.

In contrast, *Mosaic* can perform coalescing in-place, i.e., base pages do *not* need to be migrated in order to be coalesced into a large page. Hence, we call the page size selection mechanism of *Mosaic* the *In-Place Coalescer*. As shown in Figure 6b, the *In-Place Coalescer* causes much less DRAM channel utilization and no SM stalls, saving significant waste compared to state-of-the-art memory managers. We describe how the *In-Place Coalescer* (1) decides which pages to coalesce, and (2) updates the page table for pages that are coalesced.

Deciding When to Coalesce. Unlike existing memory managers, *Mosaic* does not need to monitor base page utilization information to identify opportunities for coalescing. Instead, we design CoCoA to ensure that the base pages that we coalesce are already allocated to the same large page frame. Once CoCoA has allocated data within a large page frame, it sends the address of the frame to the *In-Place Coalescer*. The *In-Place Coalescer* then checks to see whether the base pages within the frame are contiguous in both virtual and physical memory.[2] As mentioned in Section 4.2, *Mosaic* coalesces base pages into a large page only if all of the base pages within the large page frame are allocated (i.e., the frame is fully populated). We empirically find that for GPGPU applications, coalescing only contiguous base pages in fully-populated large page frames achieves similar TLB reach to the coalescing performed by existing memory managers (not shown), and avoids the need to employ an MMU or perform page migration, which greatly reduces the overhead of *Mosaic*.

Coalescing in Hardware. Once the *In-Place Coalescer* selects a large page frame for coalescing, it then performs the coalescing operation in hardware. Figure 6b shows the steps required for coalescing with the *In-Place Coalescer* under the *Mosaic* timeline. Unlike coalescing in existing memory managers, the *In-Place Coalescer* does *not* need to perform any data migration, as CoCoA has already conserved contiguity within all large page frames selected for coalescing: the coalescing operation needs to only update the page table entries corresponding to the large page frame and the base pages (❹ in the figure).

We modify the L3 and L4 page table entries (PTEs) to simplify updates during the coalescing operation, as shown in Figure 7a. We add a *large page bit* to each L3 PTE (corresponding to a large page), which is initially set to 0 (to indicate a page that is *not* coalesced), and we add a *disabled bit* to each L4 PTE (corresponding to a base page), which is initially set to 0 (to indicate that a page table walker should use the base page virtual-to-physical mapping in the L4 PTE). The coalescing hardware simply needs to locate the L3 PTE for the large page frame being coalesced (❶ in the figure), and set the *large page* bit to 1 for the PTE (❷). (We discuss how page table lookups occur below.) We perform this bit setting operation atomically, with a single memory operation to minimize the amount of time before the large page mapping can be used. Then, the coalescing hardware sets the *disabled* bit to 1 for all L4 PTEs (❸).

The virtual-to-physical mapping for the large page can be used as soon as the large page bit is set, without (1) waiting for the disabled bits in the L4 PTEs to be set, or (2) requiring a TLB flush

[2] Coalescing decisions are made purely in the software runtime portion of the *In-Place Coalescer*, and thus system designers can easily use a different coalescing policy, if desired.

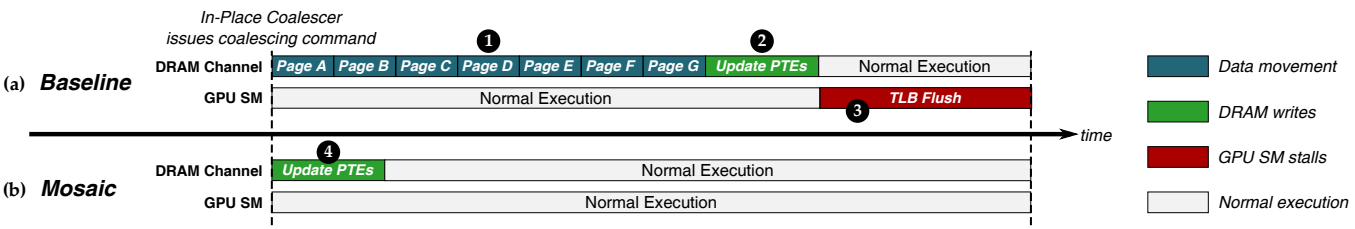
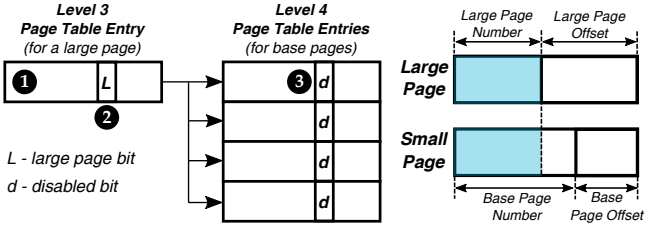


Figure 6: Coalescing timeline for (a) GPU-MMU baseline and for (b) *Mosaic*.



(a) Page table entries. **(b) Virtual-to-physical mappings in an L4 PTE.**

Figure 7: L3 and L4 page table structure in *Mosaic*.

to remove the base page mappings from the TLB. This is because no migration was needed to coalesce the base pages into the large page. As a result, the existing virtual-to-physical mappings for the coalesced base pages still point to the correct memory locations. While we set the disabled bits in the PTEs to discourage using these mappings, as the mappings consume a portion of the limited number of base page entries in the TLB, we can continue to use the mappings *safely* until they are evicted from the TLB. As shown in Figure 6b, since we do not flush the TLB, we do not need to stall the SMs. *Mosaic* ensures that if the coalesced page is subsequently splintered, the large page virtual-to-physical mapping is removed (see Section 4.4).[3]

As we can see from Figure 6, the lack of data migration and TLB flushes in *Mosaic* greatly reduces the time required for the coalescing operation in *Mosaic*, with respect to coalescing in existing MMUs.

TLB Lookups After Coalescing. As mentioned in Section 2.2, each TLB level contains two separate sets of entries, with one set of entries for each page size. In order to improve TLB reach, we need to ensure that an SM does not fetch the *base* page PTEs for coalesced base pages (even though these are safe to use) into the TLBs, as these PTEs contend with PTEs of uncoalesced base pages for the limited TLB space. When a GPU with *Mosaic* needs to translate a memory address, it first checks if the address belongs to a coalesced page by looking up the TLB large page entries. If the SM locates a valid large page entry for the request (i.e., the page is coalesced), it avoids looking up TLB base page entries.

If a TLB miss occurs in both the TLB large page and base page entries for a coalesced page, the page walker traverses the page table. At the L3 PTE (❶ in Figure 7a), the walker reads the large page bit (❷). As the bit is set, the walker needs to read the virtual-to-physical mapping for the large page. The L3 PTE does *not* typically contain space for a virtual-to-physical mapping, so the walker instead reads the virtual-to-physical mapping from the first PTE of the L4 page table that the L3 PTE points to. Figure 7b shows why we can use the mapping in the L4 PTE for the large page. A virtual-to-physical mapping for a large page consists of a page number and an offset. As the base pages within the large page were not migrated, their mappings point to physical memory locations within the large page frame. As a result, if we look at only the bits of the mapping used for the large page number, they are identical for both the large page mapping and the base page mapping. When the large page bit is set, the page walker reads the large page number from the L4 PTE (along with other fields of the PTE, e.g., for access permissions), and returns the PTE to the TLB. In doing so, we do not need to allocate any extra storage for the virtual-to-physical mapping of the large page. Note that for pages that are *not* coalesced, the page walker behavior is *not* modified.

[3] As there is a chance that base pages within a splintered page can migrated during compaction, the large page virtual-to-physical mapping may no longer be valid. To avoid correctness issues when this happens, *Mosaic* flushes the TLB large page entry for the mapping as soon as a coalesced page is splintered.

4.4 Contiguity-Aware Compaction

After an application kernel finishes, it can deallocate some of the base pages that it previously allocated. This deallocation can lead to internal fragmentation within a large page frame that was coalesced, as some of the frame's constituent base pages are no longer valid. While the page could still benefit from coalescing (as this improves TLB reach), the unallocated base pages within the large page frame cannot be allocated to another virtual address as long as the page remains coalesced. If significant memory fragmentation exists, this can cause CoCoA to run out of free large page frames, even though it has not allocated all of the available base pages in GPU memory. To avoid an out-of-memory error in the application, *Mosaic* uses CAC to splinter and compact highly-fragmented large page frames, freeing up large page frames for CoCoA to use.

Deciding When to Splinter and Compact a Coalesced Page. Whenever an application deallocates a base page within a coalesced large page frame, CAC checks to see how many base pages remain allocated within the frame. If the number of allocated base pages falls below a predetermined threshold (which is configurable in the GPU runtime), CAC decides to splinter the large page frame into base pages (see below). Once the splintering operation completes, CACperforms compaction by migrating the remaining base pages to another uncoalesced large page frame that belongs to the same application. In order to avoid occupying multiple memory channels while performing this migration, which can hurt the performance of other threads that are executing concurrently, we restrict CAC to migrate base pages between only large page frames that reside within the *same* memory channel. After the migration is complete, the original large page frame no longer contains any allocated base pages, and CAC sends the address of the large page frame to CoCoA, which adds the address to its free frame list.

If the number of allocated base pages within a coalesced page is greater than or equal to the threshold, CAC does *not* splinter the page, but notifies CoCoA of the large page frame address. CoCoA then stores the coalesced large page frame's address in a *emergency frame list*. As a failsafe, if CoCoA runs out of free large pages, and CAC does not have any large pages that it can compact, CoCoA pulls a coalesced page from the emergency frame list, asks CAC to splinter the page, and then uses any unallocated base pages within the splintered large page frame to allocate new virtual base pages.

Splintering the Page in Hardware. Similar to the *In-Place Coalescer*, when CAC selects a coalesced page for splintering, it then performs the splintering operation in hardware. The splintering operation essentially reverses the coalescing operation. First, the splintering hardware clears the disabled bit in the L4 PTEs of the constituent base pages. Then, the splintering hardware clears the large page bit atomically, which causes the subsequent page table walks to look up the virtual-to-physical mapping for the base page. Unlike coalescing, when the hardware splinters a coalesced page, it must also issue a TLB flush request for the coalesced page. As we discuss in Section 4.3, a large page mapping can be present in the TLB only when a page is coalesced. The flush to the TLB removes the large page entry for this mapping, to ensure synchronization across all SMs with the current state of the page table.

Optimizing Compaction with Bulk Copy Mechanisms. The migration of each base page during compaction requires several long-latency memory operations, where the contents of the page are copied to a destination location only 64 bits at a time, due to

the narrow width of the memory channel [61, 101, 102]. To optimize the performance of CAC, we can take advantage of in-DRAM bulk copy techniques such as RowClone [101, 102] or LISA [21], which provide very low-latency (e.g., 80 ns) memory copy within a single DRAM module. These mechanisms use existing internal buses within DRAM to copy an entire base page of memory with a single bulk memory operation. While such bulk data copy mechanisms are not essential for our proposal, they have the potential to improve performance when a large amount of compaction takes place. Section 6.4 evaluates the benefits of using in-DRAM bulk copy with CAC.

5 METHODOLOGY

We modify the MAFIA framework [45], which uses GPGPU-Sim 3.2.2 [12], to evaluate *Mosaic* on a GPU that concurrently executes multiple applications. We have released our simulator modifications [98, 99]. Table 1 shows the system configuration we simulate for our evaluations, including the configurations of the GPU core and memory partition (see Section 2.1).

GPU Core Configuration	
Shader Core Config	30 cores, 1020 MHz, GTO warp scheduler [96]
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before accessing L2, 1-cycle latency
Private L1 TLB	128 base page/16 large page entries per core, fully associative, LRU, single port, 1-cycle latency

Memory Partition Configuration (6 memory partitions in total, with each partition accessible by all 30 cores)	
Shared L2 Cache	2MB total, 16-way associative, LRU, 2 cache banks and 2 ports per memory partition, 10-cycle latency
Shared L2 TLB	512 base page/256 large page entries, non-inclusive, 16-way/fully-associative (base page/large page), LRU, 2 ports, 10-cycle latency
DRAM	3GB GDDR5, 1674 MHz, 6 channels, 8 banks per rank, FR-FCFS scheduler [94, 119], burst length 8

Table 1: Configuration of the simulated system.

Simulator Modifications. We modify GPGPU-Sim [12] to model the behavior of Unified Virtual Address Space [77]. We add a memory allocator into cuda-sim, the CUDA simulator within GPGPU-Sim, to handle all virtual-to-physical address translations and to provide memory protection. We add an accurate model of address translation to GPGPU-Sim, including TLBs, page tables, and a page table walker. The page table walker is shared across all SMs, and allows up to 64 concurrent walks. Both the L1 and L2 TLBs have separate entries for base pages and large pages [35, 50, 51, 84, 89, 90]. Each TLB contains miss status holding registers (MSHRs) [58] to track in-flight page table walks. Our simulation infrastructure supports demand paging, by detecting page faults and faithfully modeling the system I/O bus (i.e., PCIe) latency based on measurements from NVIDIA GTX 1080 cards [83] (see Section 3.2).[4] We use a worst-case model for the performance of our compaction mechanism (CAC, see Section 4.4) conservatively, by stalling the *entire* GPU (all SMs) and flushing the pipeline. More details about our modifications can be found in our extended technical report [11].

Workloads. We evaluate the performance of *Mosaic* using both *homogeneous* and *heterogeneous* workloads. We categorize each workload based on the number of concurrently-executing applications, which ranges from one to five for our homogeneous workloads, and from two to five for our heterogeneous workloads. We form our homogeneous workloads using multiple copies of the same application. We build 27 homogeneous workloads for each category using GPGPU applications from the Parboil [109], SHOC [27], LULESH [52, 53], Rodinia [22], and CUDA SDK [76] suites. We form our heterogeneous workloads by randomly selecting a number of applications out of these 27 GPGPU applications. We build 25 heterogeneous workloads per category. Each workload has a combined working set size that ranges from 10MB to 2GB. The average working set size of a workload is 217MB. In total we evaluate 235 homogeneous and heterogeneous workloads. We provide a list of all our workloads in our extended technical report [11].

Evaluation Metrics. We report workload performance using the weighted speedup metric [31, 32], which is a commonly-used metric to evaluate the performance of a multi-application workload [9, 28, 29, 54, 56, 57, 70, 71, 72]. Weighted speedup is calculated as:

$$\text{Weighted Speedup} = \sum \frac{IPC_{shared}}{IPC_{alone}} \quad (1)$$

where IPC_{alone} is the IPC of an application in the workload that runs on the same number of shader cores using the baseline state-of-the-art configuration [92], but does not share GPU resources with any other applications; and IPC_{shared} is the IPC of the application when it runs concurrently with other applications. We report the performance of each application within a workload using IPC.

Scheduling and Partitioning of Cores. As scheduling is not the focus of this work, we assume that SMs are equally partitioned across the applications within a workload, and use the greedy-then-oldest (GTO) warp scheduler [96]. We speculate that if we use other scheduling or partitioning policies, *Mosaic* would still increase the TLB reach and achieve the benefits of demand paging effectively, though we leave such studies for future work.

6 EVALUATION

In this section, we evaluate how *Mosaic* improves the performance of homogeneous and heterogeneous workloads (see Section 5 for more detail). We compare *Mosaic* to two mechanisms: (1) *GPU-MMU*, a baseline GPU with a state-of-the-art memory manager based on the work by Power et al. [92], which we explain in detail in Section 3.1; and (2) *Ideal TLB*, a GPU with an ideal TLB, where every address translation request hits in the L1 TLB (i.e., there are no TLB misses).

6.1 Homogeneous Workloads

Figure 8 shows the performance of *Mosaic*for the homogeneous workloads. We make two observations from the figure. First, we observe that *Mosaic* is able to recover most of the performance lost due to the overhead of address translation (i.e., an ideal TLB) in homogeneous workloads. Compared to the GPU-MMU baseline, *Mosaic* improves the performance by 55.5%, averaged across all 135 of our homogeneous workloads. The performance of *Mosaic* comes within 6.8% of the *Ideal TLB* performance, indicating that *Mosaic* is effective at extending the TLB reach. Second, we observe that *Mosaic* provides good scalability. As we increase the number of concurrently-executing applications, we observe that the performance of *Mosaic* remains close to the *Ideal TLB* performance.

We conclude that for homogeneous workloads, *Mosaic* effectively approaches the performance of a GPU with the *Ideal TLB*, by employing multiple page sizes to simultaneously increase the reach of both the L1 private TLB and the shared L2 TLB.

[4] Our experience with the NVIDIA GTX 1080 suggests that production GPUs perform significant prefetching to reduce latencies when reference patterns are predictable. This feature is not modeled in our simulations.

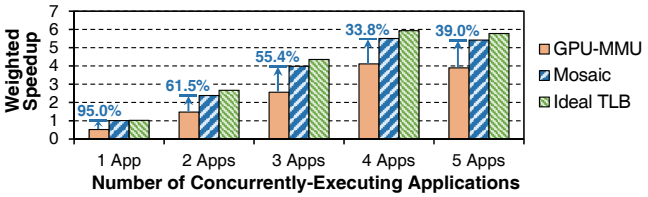


Figure 8: Homogeneous workload performance of the GPU memory managers as we vary the number of concurrently-executing applications in each workload.

6.2 Heterogeneous Workloads

Figure 9 shows the performance of *Mosaic* for heterogeneous workloads that consist of multiple randomly-selected GPGPU applications. From the figure, we observe that on average across all of the workloads, *Mosaic* provides a performance improvement of 29.7% over GPU-MMU, and comes within 15.4% of the *Ideal TLB* performance. We find that the improvement comes from the significant reduction in the TLB miss rate with *Mosaic*, as we discuss below.

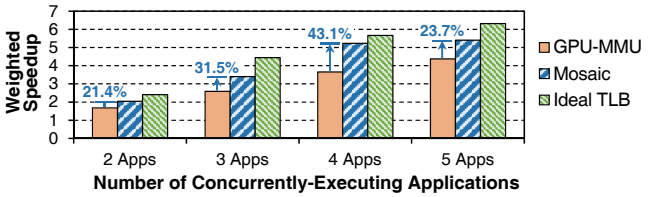


Figure 9: Heterogeneous workload performance of the GPU memory managers.

The performance gap between *Mosaic* and *Ideal TLB* is greater for heterogeneous workloads than it is for homogeneous workloads. To understand why, we examine the performance of the each workload in greater detail. Figure 10 shows the performance improvement of 15 randomly-selected two-application workloads. We categorize the workloads as either *TLB-friendly* or *TLB-sensitive*. The majority of the workloads are TLB-friendly, which means that they benefit from utilizing large pages. The TLB hit rate increases significantly with *Mosaic* (see Section 6.3) for TLB-friendly workloads, allowing the workload performance to approach *Ideal TLB*. However, for TLB-sensitive workloads, such as HS–CONS and NW–HISTO, there is still a performance gap between *Mosaic* and the *Ideal TLB*, even though *Mosaic* improves the TLB hit rate. We discover two main factors that lead to this performance gap. First, in these workloads, one of the applications is highly sensitive to shared L2 TLB misses (e.g., HS in HS–CONS, HISTO in NW–HISTO), while the other application (e.g., CONS, NW) is memory intensive. The memory-intensive application introduces a high number of conflict misses on the shared L2 TLB, which harms the performance of the TLB-sensitive application significantly, and causes the workload's performance under *Mosaic* to drop significantly below the *Ideal TLB* performance. Second, the high latency of page walks due to compulsory TLB misses and higher access latency to the shared L2 TLB (which increases because TLB requests have to probe both the large page and base page TLBs) have a high impact on the TLB-sensitive application. Hence, for these workloads, the *Ideal TLB* still has significant advantages over *Mosaic*.

Summary of Impact on Individual Applications. To determine how *Mosaic* affects the individual applications within the heterogeneous workloads we evaluate, we study the IPC of each application in all of our heterogeneous workloads. In all, this represents a total of 350 individual applications. Figure 11 shows the

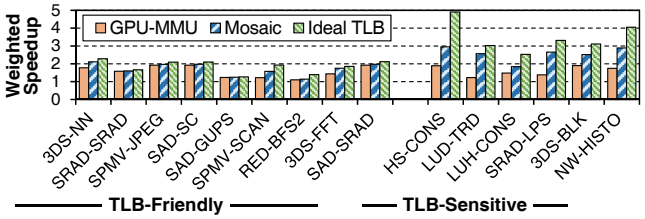


Figure 10: Performance of selected two-application heterogeneous workloads.

per-application IPC of *Mosaic* and *Ideal TLB* normalized to the application's performance under GPU-MMU, and sorted in ascending order. We show four graphs in the figure, where each graph corresponds to individual applications from workloads with the same number of concurrently-executing applications. We make three observations from these results. First, *Mosaic* improves performance relative to GPU-MMU for 93.6% of the 350 individual applications. We find that the application IPC relative to the baseline GPU-MMU for each application ranges from 66.3% to 860%, with an average of 133.0%. Second, for the 6.4% of the applications where *Mosaic* performs worse than GPU-MMU, we find that for each application, the other concurrently-executing applications in the same workload experience a significant performance improvement. For example, the worst-performing application, for which *Mosaic* hurts performance by 33.6% compared to GPU-MMU, is from a workload with three concurrently-executing applications. We find that the other two applications perform 66.3% and 7.8% better under *Mosaic*, compared to GPU-MMU. Third, we find that, on average across all heterogeneous workloads, 48%, 68.9% and 82.3% of the applications perform within 90%, 80% and 70% of *Ideal TLB*, respectively.

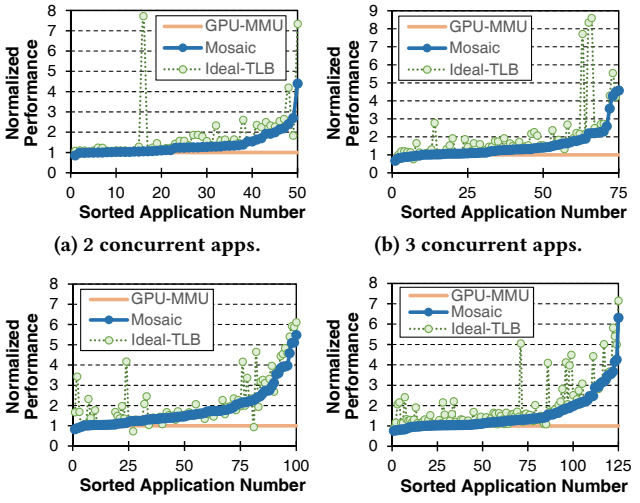


Figure 11: Sorted normalized per-application IPC for applications in heterogeneous workloads, categorized by the number of applications in a workload.

We conclude that *Mosaic* is effective at increasing the TLB reach for heterogeneous workloads, and delivers significant performance improvements over a state-of-the-art GPU memory manager.

Impact of Demand Paging on Performance. All of our results so far show the performance of the GPU-MMU baseline and *Mosaic* when demand paging is *enabled*. Figure 12 shows the normalized

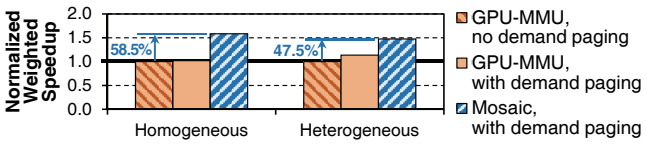


Figure 12: Performance of GPU-MMU and *Mosaic* compared to GPU-MMU without demand paging.

weighted speedup of the GPU-MMU baseline and *Mosaic*, compared to GPU-MMU *without demand paging*, where all data required by an application is moved to the GPU memory *before* the application starts executing. We make two observations from the figure. First, we find that *Mosaic* outperforms GPU-MMU without demand paging by 58.5% on average for homogeneous workloads and 47.5% on average for heterogeneous workloads. Second, we find that demand paging has little impact on the weighted speedup. This is because demand paging latency occurs only when a kernel launches, at which point the GPU retrieves data from the CPU memory. The data transfer overhead is required regardless of whether demand paging is enabled, and thus the GPU incurs similar overhead with and without demand paging.

6.3 Analysis of TLB Impact

TLB Hit Rate. Figure 13 compares the overall TLB hit rate of GPU-MMU to *Mosaic* for 214 of our 235 workloads, which suffer from *limited TLB reach* (i.e., workloads that have an L2 TLB hit rate lower than 98%). We make two observations from the figure. First, we observe *Mosaic* is very effective at increasing the TLB reach of these workloads. We find that for the GPU-MMU baseline, *every* fully-mapped large page frame contains pages from multiple applications, as the GPU-MMU allocator does not provide the soft guarantee of CoCoA. As a result, GPU-MMU does not have any opportunities to coalesce base pages into a large page without performing significant amounts of data migration. In contrast, *Mosaic* can coalesce a vast majority of base pages thanks to CoCoA. As a result, *Mosaic* reduces the TLB miss rate dramatically for these workloads, with the average miss rate falling below 1% in both the L1 and L2 TLBs. Second, we observe an increasing amount of interference in GPU-MMU when more than three applications are running concurrently. This results in a lower TLB hit rate as the number of applications increases from three to four applications, and from four to five applications. The L2 TLB hit rate drops from 81% in workloads with two concurrently-executing applications to 62% in workloads with five concurrently-executing applications. *Mosaic* experiences no such drop due to interference as we increase the number of concurrently-executing applications, since it makes much greater use of large page coalescing and enables a much larger TLB reach.

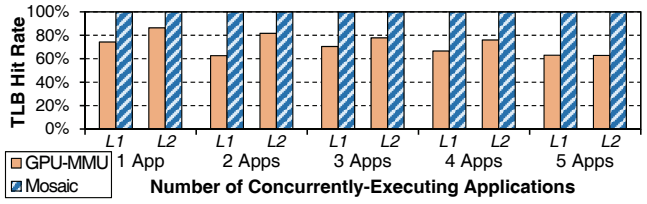


Figure 13: L1 and L2 TLB hit rate for GPU-MMU and *Mosaic*.

TLB Size Sensitivity. A major benefit of *Mosaic* is its ability to improve TLB reach by increasing opportunities to coalesce base pages into a large page. After the base pages are coalesced, the GPU uses the large page TLB to cache the virtual-to-physical mapping of the large page, which frees up base page TLB entries so that they can be used to cache mappings for the uncoalesced base pages. We now evaluate how sensitive the performance of *Mosaic* is to the number of base page and large page entries in each TLB level.

Figure 14 shows the performance of both GPU-MMU and *Mosaic* as we vary the number of *base page entries* in the per-SM L1 TLBs (Figure 14a) and in the shared L2 TLB (Figure 14b). We normalize all results to the GPU-MMU performance with the baseline 128-base-page-entry L1 TLBs per SM and a 512-base-page-entry shared L2 TLB. From the figure, we make two observations. First, we find that for the L1 TLB, GPU-MMU is sensitive to the number of base page entries, while *Mosaic* is *not sensitive* to the number of base page entries. This is because *Mosaic* successfully coalesces most of its base pages into large pages, which significantly reduces the pressure on TLB base page capacity. In fact, the number of L1 TLB base page entries has a minimal impact on the performance of *Mosaic* until we scale it all the way down to 8 entries. Even then, compared to an L1 TLB with 128 base page entries, *Mosaic* loses only 7.6% performance on average with 8 entries. In contrast, we find that GPU-MMU is unable to coalesce base pages, and as a result, its performance scales poorly as we reduce the number of TLB base page entries. Second, we find that the performance of both GPU-MMU and *Mosaic* is sensitive to the number of L2 TLB base page entries. This is because even though *Mosaic* does not need many L1 TLB base page entries per SM, the base pages are often **shared** across multiple SMs. The L2 TLB allows SMs to share page table entries (PTEs) with each other, so that once an SM retrieves a PTE from memory using a page walk, the other SMs do not need to wait on a page walk. The larger the number of L2 TLB base page entries, the more likely it is that a TLB request can avoid the need for a page walk. Since *Mosaic* does not directly have an effect on the number of page walks, it benefits from a mechanism (e.g., a large L2 TLB) that can reduce the number of page walks and hence is sensitive to the size of the L2 TLB.

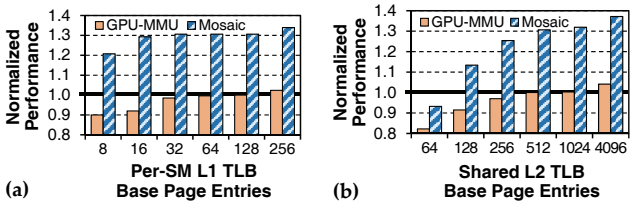


Figure 14: Sensitivity of GPU-MMU and *Mosaic* performance to L1 and L2 TLB base page entries, normalized to GPU-MMU with 128 L1 and 512 L2 TLB base page entries.

Figure 15 shows the performance of both GPU-MMU and *Mosaic* as we vary the number of *large page entries* in the per-SM L1 TLBs (Figure 15a) and in the shared L2 TLB (Figure 15b). We normalize all results to the GPU-MMU performance with the baseline 16-large-page-entry L1 TLBs per SM and a 256-large-page-entry shared L2 TLB. We make two observations from the figure. First, for both the L1 and L2 TLBs, *Mosaic* is sensitive to the number of large page entries. This is because *Mosaic* successfully coalesces most of its base pages into large pages. We note that the sensitivity is not as high as *Mosaic*'s sensitivity to L2 TLB base page entries (Figure 14b), because each large page entry covers a much larger portion of memory, which allows a smaller number of large page entries to still cover a majority of the total application memory. Second, GPU-MMU is insensitive to the large page entry count. This is because GPU-MMU is unable to coalesce any base pages into large pages, due to its coalescing-unfriendly allocation (see

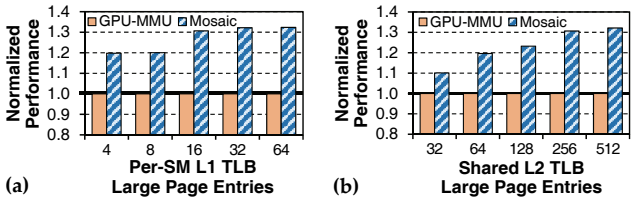


Figure 15: Sensitivity of GPU-MMU and *Mosaic* performance to L1 and L2 TLB large page entries, normalized to GPU-MMU with 16 L1 and 256 L2 TLB large page entries.

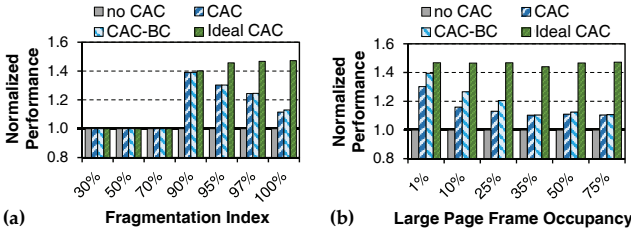


Figure 16: Performance of CAC under varying degrees of (a) fragmentation and (b) large page frame occupancy.

Figure 1a). As a result, GPU-MMU makes *no use* of the large page entries in the TLB.

6.4 Analysis of the Effect of Fragmentation

When multiple concurrently-executing GPGPU applications share the GPU, a series of memory allocation and deallocation requests could create significant data fragmentation, and could cause CoCoA to violate its soft guarantee, as discussed in Section 4.2. While we do not observe this behavior in any of the workloads that we evaluate, *Mosaic* can potentially introduce data fragmentation and memory bloat for very long running applications. In this section, we design stress-test experiments that induce a large amount of fragmentation in large page frames, to study the behavior of CoCoA and CAC.

To induce a large amount of fragmentation, we allow the memory allocator to *pre-fragment* a fraction of the main memory. We randomly place pre-fragmented data throughout the physical memory. This data (1) does not conform to *Mosaic*'s soft guarantee, and (2) cannot be coalesced with any other base pages within the same large page frame. To vary the degree of large page fragmentation, we define two metrics: (1) the *fragmentation index*, which is the fraction of large page frames that contain pre-fragmented data; and (2) *large page frame occupancy*, which is the fraction of the pre-fragmented data that occupies each fragmented large page.

We evaluate the performance of all our workloads on (1) *Mosaic* with the baseline CAC; and (2) *Mosaic* with an optimized CAC that takes advantage of in-DRAM bulk copy mechanisms (see Section 4.4), which we call *CAC-BC*. We provide a comparison against two configurations: (1) *Ideal CAC*, a compaction mechanism where data migration incurs zero latency; and (2) *No CAC*, where CAC is not applied.

Figure 16a shows the performance of CAC when we vary the fragmentation index. For these experiments, we set the large page frame occupancy to 50%. We make three observations from Figure 16a. First, we observe that there is minimal performance impact when the fragmentation index is less than 90%, indicating that it is unnecessary to apply CAC unless the main memory is heavily fragmented. Second, as we increase the fragmentation index above 90%, CAC provides performance improvements for *Mosaic*, as CAC effectively frees up large page frames and prevents CoCoA from running out of frames. Third, we observe that as the fragmentation index approaches 100%, CAC becomes less effective, due to the fact that compaction needs to be performed very frequently, causing a significant amount of data migration.

Figure 16b shows the performance of CAC as the large page frame occupancy changes when we set the fragmentation index to 100% (i.e., *every* large page frame is pre-fragmented). We make two observations from the figure. First, we observe that CAC-BC is effective when occupancy is no greater than 25%. When the occupancy is low, in-DRAM bulk-copy operations can effectively reduce the overhead of CAC, as there are many opportunities to free up large page frames that require data migration. Second, we observe that as the occupancy increases beyond 35% (i.e., many base pages are already allocated), the benefits of CAC and CAC-BC decrease, as (1) fewer large page frames can be freed up by compaction, and (2) more base pages need to be moved in order to free a large page frame.

Table 2 shows how CAC controls memory bloat for different large page frame occupancies, when we set the fragmentation index to 100%. When large page frames are used, memory bloat can increase as a result of high fragmentation. We observe that when pages are aggressively pre-fragmented, CAC is effective at reducing the memory bloat resulting from high levels of fragmentation. For example, when the large page frame occupancy is very high (e.g., above 75%), CAC compacts the pages effectively, reducing memory bloat to within 2.2% of the memory that would be allocated if we were to use only 4KB pages (i.e., when no large page fragmentation exists). We observe negligible (<1%) memory bloat when the fragmentation index is less than 100% (not shown), indicating that CAC is effective at mitigating large page fragmentation.

Large Page Frame Occupancy (%)	1%	10%	25%	35%	50%	75%
Memory Bloat	10.66%	7.56%	7.20%	5.22%	3.37%	2.22%

Table 2: Memory bloat of *Mosaic*, compared to a GPU-MMU memory manager that uses only 4KB base pages.

We conclude that CoCoA and CAC work together effectively to preserve virtual and physical address contiguity within a large page frame, without incurring high data migration overhead and memory bloat.

7 RELATED WORK

To our knowledge, this is the first work to (1) analyze the fundamental trade-offs between TLB reach, demand paging performance, and internal page fragmentation; and (2) propose an application-transparent GPU memory manager that preemptively coalesces pages at allocation time to improve address translation performance, while avoiding the demand paging inefficiencies and memory copy overheads typically associated with large page support. Reducing performance degradation from address translation overhead is an active area of work for CPUs, and the performance loss that we observe as a result of address translation is well corroborated [15, 17, 34, 36, 67]. In this section, we discuss previous techniques that aim to reduce the overhead of address translation.

7.1 TLB Designs for CPU Systems

TLB miss overhead can be reduced by (1) accelerating page table walks [13, 16] or reducing the walk frequency [35]; or (2) reducing

the number of TLB misses (e.g., through prefetching [18, 49, 100], prediction [84], or structural changes to the TLB [88, 89, 111] or TLB hierarchy [4, 5, 15, 17, 34, 50, 64, 108]).

Support for Multiple Page Sizes. Multi-page mapping techniques [88, 89, 111] use a single TLB entry for multiple page translations, improving TLB reach by a small factor; much greater improvements to TLB reach are needed to deal with modern memory sizes. MIX TLB [26] accommodates entries that translate multiple page sizes, eliminating the need for a dedicated set of TLB large page entries. MIX TLB is orthogonal to our work, and can be used with *Mosaic* to further improve TLB reach.

Navarro et al. [74] identify contiguity-awareness and fragmentation reduction as primary concerns for large page management, proposing reservation-based allocation and deferred promotion (i.e., coalescing) of base pages to large pages. Similar ideas are widely used in modern OSes [24]. Instead of the reservation-based scheme, Ingens [59] employs a utilization-based scheme that uses a bit vector to track spatial and temporal utilization of base pages.

Techniques to Increase Memory Contiguity. GLUE [90] groups contiguous, aligned base page translations under a single speculative large page translation in the TLB. GTSM [30] provides hardware support to leverage the contiguity of physical memory region even when pages have been retired due to bit errors. These mechanisms for preserving or recovering contiguity are orthogonal to the contiguity-conserving allocation we propose for *Mosaic*, and they can help *Mosaic* by avoiding the need for compaction.

Gorman et al. [39] propose a placement policy for an OS's physical page allocator that mitigates fragmentation and promotes contiguity by grouping pages according to the amount of migration required to achieve contiguity. Subsequent work [40] proposes a software-exposed interface for applications to explicitly request large pages like `libhugetlbfs` [38]. These ideas are complementary to our work. *Mosaic* can potentially benefit from similar policies if they can be simplified enough to be implementable in hardware.

Alternative TLB Designs. Research on shared last-level TLB designs [17, 19, 64] and page walk cache designs [16] has yielded mechanisms that accelerate multithreaded CPU applications by sharing translations between cores. SpecTLB [14] provides a technique that predicts address translations. While speculation works on CPU applications, speculation for highly-parallel GPUs is more complicated, and can eventually waste off-chip DRAM bandwidth, which is a highly-contended resource in GPUs. Direct segments [15] and redundant memory mappings [50] provide virtual memory support for server workloads that reduce the overhead of address translation. These proposals map large contiguous chunks of virtual memory to the physical address space in order to reduce the address translation overhead. While these techniques improve the TLB reach, they increase the transfer latency depending on the size of the virtual chunks they map.

7.2 TLB Designs for GPU Systems

TLB Designs for Heterogeneous Systems. Previous works provide several TLB designs for heterogeneous systems with GPUs [91, 92, 114] and with accelerators [23]. *Mosaic* improves upon a state-of-the-art TLB design [92] by providing application-transparent, high-performance support for multiple page sizes in GPUs. No prior work provides such support.

TLB-Aware Warp Scheduler. Pichai et al. [91] extend the cache-conscious warp scheduler [96] to be aware of the TLB in heterogeneous CPU-GPU systems. These techniques are orthogonal to the problem we focus on, and can be applied in conjunction with *Mosaic* to further improve performance.

Analysis of Address Translation in GPUs. Vesely et al. [114] 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. They observe that high-latency address translations limit the GPU's latency hiding capability and hurt performance, which is in line with the observations we make in Section 3. Mei et al. [66] use a set of microbenchmarks to evaluate the address translation process in commercial GPUs. Their work concludes that previous NVIDIA architectures [78, 79] have *off-chip* L1 and L2 TLBs, which lead to poor performance.

Other Ways to Manage Virtual Memory. VAST [62] is a software-managed virtual memory space for GPUs. In that work, the authors observe that the limited size of physical memory typically prevents data-parallel programs from utilizing GPUs. To address this, VAST automatically partitions GPU programs into chunks that fit within the physical memory space to create an illusion of virtual memory. Unlike *Mosaic*, VAST is unable to provide memory protection from concurrently-executing GPGPU applications. Zorua [115] is a holistic mechanism to virtualize multiple hardware resources within the GPU. Zorua does not virtualize the main memory, and is thus orthogonal to our work. CABA [116] introduces assist warps, which act as background helper threads for GPU SMs. These assist warps can be adapted to perform various memory virtualization functions, such as page walks and base page utilization analysis.

7.3 Demand Paging for GPUs

Demand paging is a challenge for GPUs [114]. Recent works [3, 118], and the AMD hUMA [60] and NVIDIA PASCAL architectures [82, 118] provide various levels of support for demand paging in GPUs. These techniques do not tackle the existing trade-off in GPUs between using large pages to improve address translation and using base pages to minimize demand paging overhead, which we relax with *Mosaic*. As we discuss in Section 3, support for multiple page sizes can be adapted to minimize the overhead of demand paging by limiting demand paging to base pages only.

8 CONCLUSION

We introduce *Mosaic*, a new GPU memory manager that provides application-transparent support for multiple page sizes. The key idea of *Mosaic* is to perform demand paging using *smaller* page sizes, and then coalesce small (i.e., base) pages into a *larger* page immediately after allocation, which allows address translation to use large pages and thus increase TLB reach. We have shown that *Mosaic* significantly outperforms state-of-the-art GPU address translation designs and achieves performance close to an ideal TLB, across a wide variety of workloads. We conclude that *Mosaic* effectively combines the benefits of large pages and demand paging in GPUs, thereby breaking the conventional tension that exists between these two concepts. We hope the ideas presented in this paper can lead to future works that analyze *Mosaic* in detail and provide even lower-overhead support for synergistic address translation and demand paging in heterogeneous systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and SAFARI group members for their feedback. Special thanks to Nastaran Hajinazar, Juan Gómez Luna, and Mohammad Sadr for their feedback. We acknowledge the support of our industrial partners, especially Google, Intel, Microsoft, NVIDIA, Samsung, and VMware. This research was partially supported by the NSF (grants 1409723 and 1618563), the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

REFERENCES

[1] A. Abrevaya, "Linux Transparent Huge Pages, JEMalloc and NuoDB," 2014.
[2] Advanced Micro Devices, Inc., "OpenCL: The Future of Accelerated Application Performance Is Now," https://www.amd.com/Documents/FirePro_OpenCL_Whitepaper.pdf.
[3] 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.
[4] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-Assisted Page Walks for Virtualized Systems," in *ISCA*, 2012.
[5] J. Ahn, S. Jin, and J. Huh, "Fast Two-Level Address Translation for Virtualized Systems," *IEEE TC*, 2015.
[6] Apple Inc., "Huge Page Support in Mac OS X," http://blog.couchbase.com/often-overlooked-linux-os-tweaks, 2014.
[7] ARM Holdings, "ARM Cortex-A Series," http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf, 2015.
[8] R. Ausavarungnirun, "Techniques for Shared Resource Management in Systems with Throughput Processors," Ph.D. dissertation, Carnegie Mellon Univ., 2017.
[9] 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.
[10] R. Ausavarungnirun, S. Ghose, O. Kayıran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *PACT*, 2015.
[11] 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," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. TR-2017-003, 2017.
[12] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.
[13] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," in *ISCA*, 2010.
[14] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *ISCA*, 2011.
[15] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *ISCA*, 2013.
[16] A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," in *MICRO*, 2013.
[17] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *HPCA*, 2011.
[18] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," in *PACT*, 2009.
[19] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB for Chip Multiprocessors," in *ASPLOS*, 2010.
[20] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an Embedded Data Parallel Language," in *PPoPP*, 2011.
[21] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[22] 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.
[23] J. Cong, Z. Fang, Y. Hao, and G. Reinman, "Supporting Address Translation for Accelerator-Centric Architectures," in *HPCA*, 2017.
[24] J. Corbet, "Transparent Hugepages," https://lwn.net/Articles/359158/, 2009.
[25] Couchbase, Inc., "Often Overlooked Linux OS Tweaks," http://blog.couchbase.com/often-overlooked-linux-os-tweaks, 2014.
[26] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *ASPLOS*, 2017.
[27] 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.
[28] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.
[29] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Aérgia: Exploiting Packet Latency Slack in On-chip Networks," in *ISCA*, 2010.
[30] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting Superpages in Non-Contiguous Physical Memory," in *HPCA*, 2015.
[31] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
[32] S. Eyerman and L. Eeckhout, "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance," *IEEE CAL*, 2014.
[33] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO*, 2007.
[34] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *MICRO*, 2014.
[35] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," in *ISCA*, 2016.
[36] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema, "Large Pages May Be Harmful on NUMA Systems," in *USENIX ATC*, 2014.
[37] D. Gay and A. Aiken, "Memory Management with Explicit Regions," in *PLDI*, 1998.
[38] M. Gorman, "Huge Pages Part 2 (Interfaces)," https://lwn.net/Articles/375096/, 2010.
[39] M. Gorman and P. Healy, "Supporting Superpage Allocation Without Additional Hardware Support," in *ISMM*, 2008.
[40] M. Gorman and P. Healy, "Performance Characteristics of Explicit Superpage Support," in *WIOSCA*, 2010.
[41] 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.
[42] Intel Corp., "Intel® 64 and IA-32 Architectures Optimization Reference Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.
[43] 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.
[44] A. Jog, "Design and Analysis of Scheduling Techniques for Throughput Processors," Ph.D. dissertation, Pennsylvania State Univ., 2015.
[45] A. Jog, O. Kayıran, 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.
[46] 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.
[47] 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.
[48] 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.
[49] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-Driven Study," in *ISCA*, 2002.
[50] 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.
[51] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Unsal, "Energy-Efficient Address Translation," in *HPCA*, 2016.
[52] 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.
[53] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Lawrence Livermore National Lab, Tech. Rep. LLNL-TR-641973, 2013.
[54] O. Kayıran, N. Chidambaram, A. Jog, R. Ausavarungnirun, M. Kandemir, G. Loh, O. Mutlu, and C. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
[55] Khronos OpenCL Working Group, "The OpenCL Specification," http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf, 2008.
[56] 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.
[57] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[58] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," in *ISCA*, 1981.
[59] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *OSDI*, 2016.
[60] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," https://developer.amd.com/wordpress/media/2012/10/hsa10.pdf, Advanced Micro Devices, Inc., 2012.
[61] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[62] J. Lee, M. Samadi, and S. Mahlke, "VAST: The Illusion of a Large Memory Space for GPUs," in *PACT*, 2014.
[63] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 2008.
[64] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," *ACM TACO*, 2013.
[65] Mark Mumy, "SAP IQ and Linux Hugepages/Transparent Hugepages," http://scn.sap.com/people/markmumy/blog/2014/05/22/sap-iq-and-linux-hugepagestransparent-hugepages, SAP SE, 2014.
[66] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy Through Microbenchmarking," *IEEE TPDS*, 2017.
[67] T. Merrifield and H. R. Taheri, "Performance Implications of Extended Page Tables on Virtualized x86 Processors," in *VEE*, 2016.
[68] Microsoft Corp., *Large-Page Support in Windows*, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366720(v=vs.85).aspx.
[69] MongoDB, Inc., "Disable Transparent Huge Pages (THP)," 2017.
[70] 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.
[71] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[72] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[73] 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.
[74] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *OSDI*, 2002.
[75] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A High-Performance Big-Data-Friendly Garbage Collector," in *OSDI*, 2016.
[76] NVIDIA Corp., "CUDA C/C++ SDK Code Samples," http://developer.nvidia.com/cuda-cc-sdk-code-samples, 2011.
[77] 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.
[78] 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.
[79] NVIDIA Corp., "NVIDIA GeForce GTX 750 Ti," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014.
[80] NVIDIA Corp., "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015.
[81] NVIDIA Corp., "NVIDIA RISC-V Story," https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf, 2016.
[82] NVIDIA Corp., "NVIDIA Tesla P100," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.
[83] NVIDIA Corp., "NVIDIA GeForce GTX 1080," https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2017.
[84] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-Based Superpage-Friendly TLB Designs," in *HPCA*, 2015.
[85] PCI-SIG, "PCI Express Base Specification Revision 3.1a," 2015.
[86] 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.
[87] Peter Zaitsev, "Why TokuDB Hates Transparent HugePages," https://www.percona.com/blog/2014/07/23/why-tokudb-hates-transparent-hugepages/, Percona LLC, 2014.
[88] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *HPCA*, 2014.
[89] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *MICRO*, 2012.
[90] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Systems: Can You Have It Both Ways?" in *MICRO*, 2015.
[91] 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.
[92] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *HPCA*, 2014.
[93] Redis Labs, "Redis Latency Problems Troubleshooting," http://redis.io/topics/latency.
[94] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
[95] T. G. Rogers, "Locality and Scheduling in the Massively Multithreaded Era," Ph.D. dissertation, Univ. of British Columbia, 2015.
[96] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.
[97] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A Compiler and Runtime for Heterogeneous Systems," in *SOSP*, 2013.
[98] SAFARI Research Group, "Mosaic – GitHub Repository," https://github.com/CMU-SAFARI/Mosaic/.
[99] SAFARI Research Group, "SAFARI Software Tools – GitHub Repository," https://github.com/CMU-SAFARI/.
[100] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-Based TLB Preloading," in *ISCA*, 2000.
[101] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *ISCA*, 2013.
[102] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers*, 2017.
[103] T. Shanley, *Pentium Pro Processor System Architecture*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.
[104] R. L. Sites and R. T. Witek, *ALPHA Architecture Reference Manual*. Boston, Oxford, Melbourne: Digital Press, 1998.
[105] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *SPIE*, 1981.
[106] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," in *ICPP*, 1978.
[107] Splunk Inc., "Transparent Huge Memory Pages and Splunk Performance," http://docs.splunk.com/Documentation/Splunk/6.1.3/ReleaseNotes/SplunkandTHP, 2013.
[108] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors," in *MICRO*, 2010.
[109] 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, IMPACT Research Group, Tech. Rep. IMPACT-12-01, 2012.
[110] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-specific Languages," *TECS*, 2014.
[111] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *ASPLOS*, 1994.
[112] J. E. Thornton, "Parallel Operation in the Control Data 6600," in *AFIPS FJCC*, 1964.
[113] J. E. Thornton, *Design of a Computer–The Control Data 6600*. Scott Foresman & Co, 1970.
[114] 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.
[115] 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.
[116] 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.
[117] VoltDB, Inc., "VoltDB Documentation: Configure Memory Management," https://docs.voltdb.com/AdminGuide/adminmemmgt.php.
[118] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards High Performance Paged Memory for GPUs," in *HPCA*, 2016.
[119] 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," US Patent No. 5,630,096, 1997.