# Batch-Aware Unified Memory Management in GPUs for Irregular Workloads

Hyojong Kim
Georgia Institute of Technology

Jaewoong Sim
Intel Labs

Prasun Gera
Georgia Institute of Technology

Ramyad Hadidi
Georgia Institute of Technology

Hyesoon Kim
Georgia Institute of Technology

## Abstract

While unified virtual memory and demand paging in modern GPUs provide convenient abstractions to programmers for working with large-scale applications, they come at a significant performance cost. We provide the first comprehensive analysis of major inefficiencies that arise in page fault handling mechanisms employed in modern GPUs. To amortize the high costs in fault handling, the GPU runtime processes a large number of GPU page faults together. We observe that this batched processing of page faults introduces large-scale serialization that greatly hurts the GPU's execution throughput. We show real machine measurements that corroborate our findings.

Our goal is to mitigate these inefficiencies and enable efficient demand paging for GPUs. To this end, we propose a GPU runtime software and hardware solution that (1) increases the batch size (i.e., the number of page faults handled together), thereby amortizing the GPU runtime fault handling time, and reduces the number of batches by supporting CPU-like thread block context switching, and (2) takes page eviction off the critical path with no hardware changes by overlapping evictions with CPU-to-GPU page migrations. Our evaluation demonstrates that the proposed solution provides an average speedup of 2x over the state-of-the-art page prefetching. We show that our solution increases the batch size by 2.27x and reduces the total number of batches by 51% on average. We also show that the average batch processing time is reduced by 27%.

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

***Keywords.*** graphics processing units; unified memory management; memory oversubscription; virtual memory

**ACM Reference Format:**
Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378529

## 1 Introduction

Graphics processing units (GPUs) have been successful in providing substantial compute performance and have now become one of the major computing platforms in servers and datacenters [21, 23]. As an accelerator device, however, a conventional discrete GPU only allows access to its own device memory, so programmers need to design their applications carefully to fit in the limited capacity of the device memory. This makes it very challenging and costly to run large-scale applications with hundreds of GBs of memory footprint, such as graph computing workloads, because it requires careful data and algorithm partitioning in addition to purchasing more GPUs just for memory capacity.

To address this issue, recent GPUs support Unified Virtual Memory (UVM) [1, 38, 39]. UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging. This allows GPUs to access a page that resides in the CPU memory as if it *were* in the GPU memory, thereby allowing GPU applications to run without worrying about the device memory capacity limit. As such, UVM frees programmers from tuning an application for each individual GPU and allows the application to run on a variety of GPUs with different physical memory sizes without any source code changes. This is good for programmability and portability.

While the feature sounds promising, in reality, the benefit comes with a non-negligible performance cost. Virtual memory support requires address translation for every memory request, and its performance impact is more substantial than in CPUs because GPUs can issue a significantly larger number of memory requests in a short period of time [8, 47]. In addition, paging in and out of GPU memory requires costly

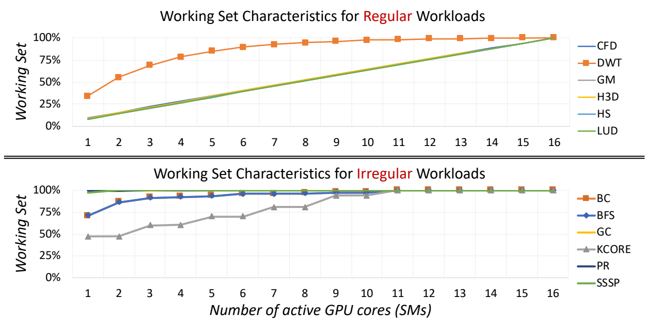

**Figure 1.** Working set size vs. active GPU core count. For most of the regular workloads, the working set size is proportional to the number of active GPU cores. In many large-scale, irregular applications, however, most memory pages are shared across GPU cores, so GPU core throttling is ineffective in reducing the working set size. All lines without markers are overlapped in each figure.

communications between CPU and GPU over an interconnect such as PCIe [44] and an interrupt handler invocation. Prior work reports that page fault handling latency ranges from 20$\mu s$ to 50$\mu s$ [53]. We find that these numbers are conservative and can be worse depending on the applications and systems. Unfortunately, this page fault latency, which is in the order of microseconds, cannot be easily hidden even with ample thread-level parallelism (TLP) in GPUs, especially when GPU memory is oversubscribed.

Recently, Li et al. [29] proposed a memory management framework, called eviction-throttling-compression (ETC), to improve GPU performance under memory oversubscription. Depending on the application characteristics, the framework selectively employs proactive eviction, memory-aware core throttling (i.e., disabling a subset of GPU cores), and capacity compression techniques. However, for many large-scale, irregular applications, we found that the ETC framework is ineffective. First, the proactive eviction heavily relies on predicting the correct timing to avoid both early and late evictions. Since irregular applications access a large number of pages within a short period of time, predicting correct timing is not trivial [29]. Second, the memory-aware throttling technique aims to reduce the application working set by disabling a subset of GPU cores. For this to be effective, the working set size has to be reduced when GPU cores are throttled. This is the case for most regular workloads, as shown in Figure 1. However, this is *not* the case for many large-scale, irregular applications because most of the memory pages are shared across GPU cores, and thus, memory-aware throttling is not effective in reducing the working set size.

The goal of this work is to support the efficient execution of large-scale irregular applications, such as graph computing workloads, in the UVM model. We first investigate how the current GPU runtime software and hardware operates for UVM (Section 2). The GPU runtime processes a *group* of GPU page faults together, rather than processing each individual one, in order to amortize the overhead of multiple round-trip latencies over the PCIe bus and to avoid invoking multiple interrupt service routines (ISRs) in the operating system (OS). To efficiently process an excessive number of page faults, the GPU runtime performs a series of operations such as preprocessing all the page faults and inserting page prefetching requests, which takes a significant amount of time (in the range of tens to hundreds of microseconds). Once all the operations (e.g., CPU page table walks for all the page faults, page allocation and eviction scheduling, etc.) are finished, page migrations between the CPU and the GPU begin. Section 3 describes our findings in detail along with the assessment of processing a group of page faults in a real GPU system.

Based on our in-depth analysis of how the GPU runtime handles GPU page faults, schedules page migrations, and interacts with the GPU hardware, we propose two novel techniques that work in synergy: (1) Thread Oversubscription (TO), a *CPU-like* thread block context switching technique, to effectively amortize the GPU runtime fault handling time by increasing the batch size (i.e., the number of page faults handled together), and (2) Unobtrusive Eviction (UE) to take GPU page evictions off the critical path with no hardware changes based on the idea of *overlapping* page evictions with CPU-to-GPU page migrations. The key contributions of this paper are as follows:

- This is the first work to discuss that a group of page faults are handled together in a batch in contemporary GPUs. We provide a comprehensive analysis of major inefficiencies that arise in the batch processing mechanism.
- Our main insight is that when page migrations account for a significant portion of the total execution time, we should consider dispatching more thread blocks to each GPU core despite the additional cost of (thread-block) context switches.
- We demonstrate that these inefficiencies can be alleviated with lightweight and practical solutions, thereby enabling more efficient demand paging for GPUs.
- We improve performance by 2x and 1.79x over the state-of-the-art page prefetching mechanism [53] and ETC mechanism [29], respectively.

## 2 Background

In this section, we first provide a brief background on GPUs in the context of thread concurrency. We then delve into the unified virtual memory feature offered by modern GPUs.

### 2.1 Thread Concurrency in GPUs

GPUs offer a high degree of data-level parallelism by executing thousands of scalar threads concurrently. To do so, a

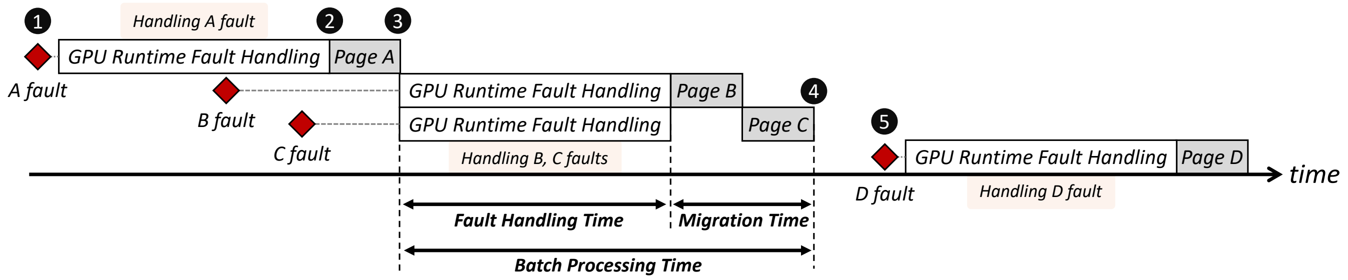

**Figure 2.** Overview of how GPU page faults are handled by the GPU runtime.

GPU shader core, such as NVIDIA Streaming Multiprocessor (SM), AMD Compute Unit (CU), or Intel Execution Unit (EU), provides hardware resources that are required to keep the contexts of multiple threads without doing conventional context switching. In each architecture, a number of factors influence thread concurrency. For example, in NVIDIA GPUs, the maximum concurrency is capped by the maximum number of threads and thread blocks (e.g., 2048 and 32, respectively), the register file size (e.g., 64k 32-bit registers), and the maximum number of registers per thread (e.g., 255), among others [38]. When a GPU kernel is launched, the GPU runtime decides the number of thread blocks to dispatch to each SM based on its hardware resources.

## 2.2 Unified Virtual Memory in GPUs

Modern GPUs offer unified virtual memory (UVM) that provides a coherent view of virtual memory address space between CPUs and GPUs in the system [1, 38, 39]. To do so, the GPU runtime software and hardware takes care of migrating pages to the memory of the accessing processor. This eliminates the need for manual page migrations, which greatly reduces the programmer's burden. It also allows us to run GPU applications that are otherwise unable to run due to memory capacity constraints. In this section, we describe in more detail how UVM works.

***Virtual Memory.*** To allow any processor in the system to access the same data, virtual memory support is required. The virtual-to-physical mapping is stored in a multi-level page table in GPUs. To translate a virtual address into a physical address, the GPU performs a page table walk. To accelerate this, translation lookaside buffers (TLBs) are adopted from CPUs and optimized for GPUs [8, 9, 46, 47]. GPUs access an order of magnitude more pages than CPUs, requiring a commensurate number of translations. In light of this, a highly threaded page table walker is proposed [47]. A multi-level page table requires many memory accesses to translate a single address. Exploiting the fact that the accesses to the upper-level page table entries have significant temporal locality, a page walk cache [10] is also adopted for GPUs [47].

***Demand Paging.*** When a GPU tries to access a physical memory page that is not currently resident in device memory, the page table walk fails. Then, the GPU generates a page fault and the GPU runtime migrates the requested page to the GPU memory. This page fault handling is expensive because (1) it requires long latency communications between the CPU and GPU over the PCIe bus, and (2) the GPU runtime performs a very expensive fault handling service routine. To amortize the overhead, the GPU runtime processes a group of page faults together, which we refer to as *batch processing*.

Figure 2 shows an overview of how GPU page faults are handled by the GPU runtime. When a page fault exception is raised by the GPU memory management unit (MMU), the GPU runtime begins to handle the exception (❶). The exception handling starts by draining all of the page fault buffer entries (page A in the figure). We use one or two pages in the figure for simplicity, but in reality, a number of page faults are generated within a short period of time since thousands of threads are concurrently running under the single-instruction multiple-threads (SIMT) execution model used in GPUs. To handle a multitude of page faults efficiently, the GPU runtime preprocesses the page faults before performing page table walks. This preprocessing includes sorting the page faults in ascending order of page addresses (to accelerate the page table walks) and the analysis of page addresses to insert page prefetching requests.[1] We refer to the time taken by the GPU runtime to perform a collection of operations to handle many page faults together as *GPU runtime fault handling* time. Specifically, the GPU runtime fault handling time is defined as the time between the beginning of batch processing and the beginning of the first page transfer to the GPU's memory by the runtime. GPU runtime fault handling time varies depending on the batch size (i.e., the number of page faults handled together in a batch) and contiguity of the pages.

The subsequent page faults generated after the batch processing begins (pages B and C in the figure) cannot be handled along with page A. Instead, they are inserted into the page fault buffer and wait until the current batch is processed

[1]Details on the preprocessing operations performed in real GPU runtime software can be found in the `preprocess_fault_batch()` function in NVIDIA driver v396.37 [41].

(❸). Once the page table walks are completed, the GPU runtime begins to migrate pages to the GPU's memory (❷). Every time a page is migrated (❸), the GPU MMU updates its page table and resumes the threads that are waiting for the page. Once the last page is migrated (❸ and ❹), the batch's processing ends. We refer to the time between the beginning of a batch's processing and the migration of the last page as *batch processing time*. When batch processing ends, the GPU runtime checks whether there are waiting page faults (pages B and C in the figure). Then, the GPU runtime begins to handle them immediately. This is an optimization to reduce the unpredictable overhead that arises due to the interrupt-based service of the OS.[2] Otherwise, the batch processing routine ends (❹). This process is repeated when a new page fault interrupt is raised by the GPU (❺).

## 3 Motivation

We make two observations on the batch processing mechanism. First, batch processing introduces a significant delay to the subsequent page fault group (or batch). As an example, take the page B fault in Figure 2, which is generated in the GPU after the first batch's processing begins. Since it cannot be handled along with page A, it has to wait until the current batch is processed. To provide perspective, we profile a breadth first search (BFS) application on an NVIDIA Titan Xp [38] GPU. For this, we use the NVIDIA Visual Profiler [42] with --unified-memory-profiling and --track-memory-allocations on. It provides timestamps for every event throughout the execution, including when the runtime starts to handle a GPU page fault group and when each page migration begins and ends. The GPU runtime fault handling time is calculated to be the duration between when it starts to handle a GPU page fault group and when it begins to migrate the first page of the group. The batch processing time can be obtained from the timestamps for the GPU page fault group event. The batch processing time is measured to be in the range of 223$\mu$s to 553$\mu$s with a median of 313$\mu$s, of which, GPU runtime fault handling accounts for an average of 46.69% of the time (measured to be in the range of 50$\mu$s to 430$\mu$s with a median of 140$\mu$s).

Fundamentally, two methods can mitigate the impact of this delay. The first method is to reduce the GPU runtime fault handling time itself by optimizing the GPU runtime software, which is beyond the scope of our work. The second method is to amortize the GPU runtime fault handling time. This can be attained by increasing the batch size (i.e., the number of page faults handled together in a batch). Figure 3 shows that per-page fault handling time decreases as the batch size increases.[3] In Section 4.1, we discuss the reason it is challenging to increase the batch size, and propose a technique to achieve it.

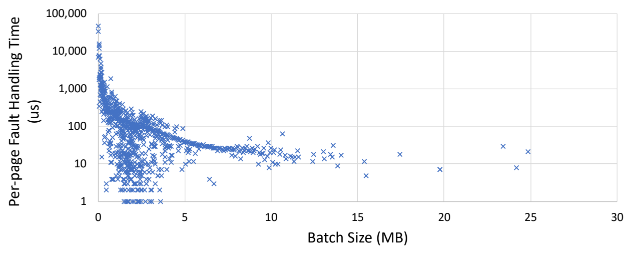

**Figure 3.** Per-page fault handling time (us) vs. batch size (MB) for breadth first search (BFS) on an NVIDIA Titan Xp GPU using the NVIDIA Visual Profiler [42].

Second, we observe that page evictions introduce unnecessary serialization in page migrations. We examine a GPU runtime implementation (NVIDIA driver v396.37 [41]) to understand how modern GPUs perform memory management and when the decision on eviction is made. When the GPU runtime handles a page fault, the physical memory allocator in the GPU runtime tries to allocate a new page (or a free root chunk) in the GPU memory (`alloc_root_chunk()`). If such an allocation fails, which indicates that the GPU memory runs out of space, a page eviction is requested (`pick_and_evict_root_chunk()`). The physical memory manager in the GPU runtime then checks whether it can evict any user memory chunks to satisfy the request.[4] Once a suitable root chunk is selected for eviction, its eviction flag is set (`chunk_start_eviction()`), and subsequently, the eviction begins (`evict_root_chunk()`). Once the eviction is completed, the metadata (e.g., tracker and status data) associated with the chunk is freed and the frame in the GPU memory becomes available. Subsequently, the new page migration begins.

From this, we conclude that page evictions and new page allocations are serialized in modern GPUs to prevent the new pages from overwriting the evicted pages. Note that an eviction is required on every page fault once the pages resident in the GPU's memory are at capacity. Figure 4 depicts these operations. When the GPU runtime fails to allocate page A, it initiates an eviction of page X, reactively (❶). Once page X is evicted from the GPU's memory, both the master page table in the CPU's memory and the GPU page table are updated for the evicted page X, and the frame is freed (❷). Once the frame is freed, page A's migration begins (❸). In Section 4.2, we propose a technique to eliminate this serialization.

[2]Details on this optimization performed in real GPU runtime software can be found in the `uvm_gpu_service_replayable_faults()` function in NVIDIA driver v396.37 [41].

[3]Fault handling time per page is calculated by dividing the batch processing time by the number of pages in the batch.

[4]All allocated user memory root chunks are tracked in an LRU list (`root_chunks.va_block_used`). A root chunk is moved to the tail of the LRU list whenever any of its sub-chunks is allocated. This is the policy referred to as aged-based LRU in literature [4, 5, 29]. To examine the head of the LRU list, `list_first_chunk()` is used.

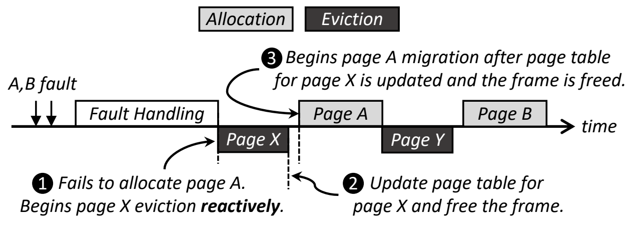
Allocation
Eviction
A,B fault
Fault Handling
Page X
Page A
Page Y
Page B
time
❸ Begins page A migration after page table for page X is updated and the frame is freed.
❶ Fails to allocate page A. Begins page X eviction **reactively**.
❷ Update page table for page X and free the frame.

**Figure 4.** Overview of how and when GPU runtime evicts a page from GPU memory, and why it is on the critical path.

## 4 Challenges and Solutions

In this section, we present two techniques to mitigate the inefficiencies described in Section 3. In Section 4.1, we propose a technique that increases the batch size to amortize the GPU runtime fault handling time and reduce the number of batches. In Section 4.2, we propose a technique that reduces the page migration time by overlapping page evictions with CPU-to-GPU page migrations without relying on precise timing of evictions.

### 4.1 Thread Oversubscription

In GPUs, the primary execution unit is a warp, which is a collection of scalar threads (e.g., 32 in NVIDIA GPUs) that run in a single-instruction multiple-thread (SIMT) fashion. A warp is stalled once it generates a page fault. Provided that only up to 64 warps (or 2048 threads) can concurrently run in an SM [38, 39], it does not take much time before the GPU becomes crippled due to lack of runnable warps. The number of concurrently running threads has been engineered to provide enough TLP to hide memory latencies in traditional GPUs, where no page migrations between the CPU and GPU occur. We find that the level of thread concurrency optimized for traditional GPUs is not sufficient to amortize the GPU runtime fault handling time in the presence of demand paging.

Two approaches can increase the batch size. The first approach is to use the stalled warps to generate more page faults. For this, runahead execution [36] or speculative execution [18, 19, 31] techniques can be employed. However, these techniques are likely less effective to generate a large number of page faults in a short amount of time because each thread block typically runs short due to the GPU programming model. The second approach is to increase thread concurrency by dispatching more thread blocks to an SM. However, the number of threads (or thread blocks) per SM is dictated by the physical resource constraint. We want a solution that can increase thread concurrency without increasing the physical resource requirement.

To this end, we develop thread oversubscription, a GPU virtualization technique. We utilize the Virtual Thread (VT) [52] as our baseline architecture for GPU virtualization. The VT architecture assigns thread blocks up to the capacity limit (i.e., physical resource constraints, such as register file and shared memory), while ignoring the scheduling limit (i.e., scheduler resource constraints, such as program counters and SIMT stacks). It dispatches thread blocks in active and inactive states, such that the number of active thread blocks does not exceed the scheduling limit. Once all the warps in an active thread block are descheduled due to long latency operations, the active thread block is context switched, and the next ready, but inactive, thread block takes its place. Since both active and inactive thread blocks fit within the capacity limit, the need to save and restore large amounts of contexts (e.g., register files) is obviated.

Our primary objective is to increase the batch size to amortize the impact of batch processing rather than just to increase TLP. However, we found that baseline VT is not applicable to most of our evaluated graph workloads as is because not enough resources are available to host even a single additional thread block. The reason is that the number of thread blocks that can be scheduled to an SM is often limited by the maximum number of threads per SM for most graph workloads. When the maximum number of threads per SM is scheduled to an SM, the maximum number of registers per SM is easily exhausted. Take NVIDIA Titan Xp [38] as an example. If the maximum number of threads per SM is 2048 and the maximum number of registers per SM is 65536 [38], each thread can use up to 32 registers. If each thread uses more than 16 registers (i.e., a half of 32 registers), which is the case for most of our evaluated workloads, baseline VT cannot host even a single additional thread block due to the register file resource constraints.

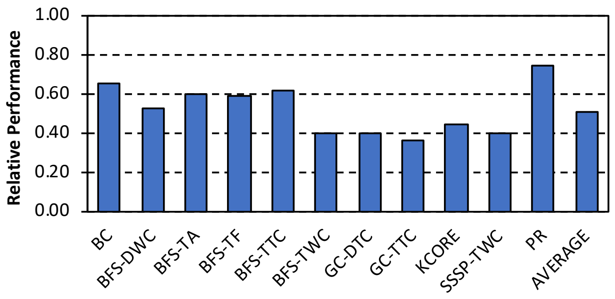

**Figure 5.** Performance degradation when provisioning an additional thread block to each SM requires context switching in traditional GPUs.

Hosting an additional (or more) thread blocks in an SM in this case requires more expensive thread-block context switching [30, 43, 49, 52]. This includes saving and restoring the per-thread-block state information (e.g., warp identifiers, thread block identifiers, and SIMT stack including the program counter) and register files to the global memory. Note that using shared memory to store the context information is no longer feasible.[5] This cost is intolerable in the traditional GPU computing model (i.e., when no page migrations between CPU and GPU memory occur). Figure 5 shows the performance impact assuming we provision an additional thread block to each SM with context switching in traditional GPUs. We see that the context switching overhead leads to a non-negligible performance degradation across all the evaluated workloads (49% on average). This indicates that the context switching overhead caused by adding an additional thread block to each SM outweighs the benefit of increasing thread concurrency if the running workloads fit in the GPU memory.

However, we observe that in the presence of page migrations between CPU and GPU memory, increasing thread concurrency is beneficial despite the expensive context switching overhead. To this end, we extend VT in three ways. First, we employ an additional mapping table so that different Virtual Warp IDs (VWIs) can access the same set of register files when they are context switched. Note that VWI is a unique warp identifier across all the assigned warps to an SM, including both active and inactive thread blocks [52]. Only when a thread block finishes execution are its VWIs released and reused for another thread block. Second, we extend the operation performed by the Virtual Thread Controller (VTC). The VTC keeps track of the state of all thread blocks in order to determine which thread blocks can be brought back from the inactive to an active state, or vice versa, when a thread block is swapped out. Baseline VT only stores the per-thread-block state information in the shared memory through the context handler. We extend this operation to store register files as well. Since the register files that a thread block uses can easily exceed tens of KBs, we use global memory instead of shared memory.

Third, we dynamically control the degree of thread oversubscription based on the rate at which premature eviction occurs. Premature eviction occurs when a page is evicted earlier than it should be, and a page fault is generated for the page again by the GPU. Since thread oversubscription increases the number of concurrently running threads, it may lead to an increase in the working set size, which increases the likelihood of premature evictions. This is detrimental because when premature eviction occurs, the evicted page has to be brought back to the GPU memory. On the other hand, an increase in thread concurrency can lead to better page utilization, reducing premature evictions. Hence, we modify the GPU runtime to monitor premature eviction rates and dynamically control the degree of thread oversubscription.

[5]Assume each thread block consists of 2048 threads and each thread uses 10 32-bit registers. In this case, 85KB = 80KB (2048 * 10 * 4 bytes) for register files + 5KB for thread block state information has to be stored and restored for context switching. The size of the thread block state information is estimated according to [52]. According to [38, 39], shared memory size can be configured up to 64KB per SM. Therefore, it is infeasible to use shared memory for context switching.

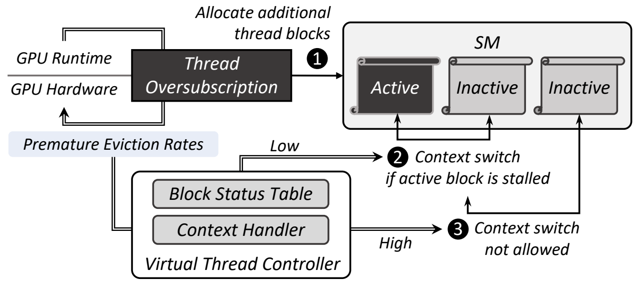

**Figure 6.** Thread oversubscription scheme.

In Section 6.1, we provide a detailed analysis of the impact on premature eviction that the thread oversubscription causes.

Figure 6 shows how our thread oversubscription technique works. We enable thread oversubscription from the beginning of the execution by allocating one additional thread block to each SM (❶). The thread block additionally allocated to each SM is inactive at first. It is important to note that the number of active thread blocks does not exceed that of the baseline, which is determined by the physical resource constraints. Once all of the warps in an active thread block are stalled due to page faults, the thread oversubscription mechanism context switches the active (but stalled) thread block with an inactive thread block (❷). The thread oversubscription mechanism can be detrimental if it causes premature evictions. To prevent this, the GPU runtime monitors the premature eviction rates by periodically estimating the running average of the lifetime of pages by tracking when each page is allocated and evicted. We use the running average as an indicator of premature evictions. If the running average is decreased by a certain threshold, the thread oversubscription mechanism does not allow any more context switching by decrementing (and limiting) the number of concurrently runnable thread blocks (❸).[6] Otherwise, thread oversubscription allocates one additional thread block to each SM in an incremental manner.

Figure 7 demonstrates how our thread oversubscription technique can increase the batch size. For ease of explanation, we assume that up to one thread block (TB) can be dispatched to an SM. We also assume that pages A, B, and C are accessed by TB1, and page D is accessed by TB2. In the baseline, TB2 can be executed only after TB1 is retired. With thread oversubscription, TB1 is context switched with TB2 when all of its warps are stalled (after a page fault for page C is generated). After the context switching overhead, TB2 is executed and a page fault for page D is generated, which can be handled along with those for pages B and C. Once page C is migrated, TB1 can be resumed and retired. Once page D is migrated, TB2 is context switched in, resumed, and retired.

[6]The number of concurrently runnable thread blocks is set to the number of allocated thread blocks at first. This does not mean all of them are running simultaneously. The warp scheduler picks a warp from active thread blocks only.

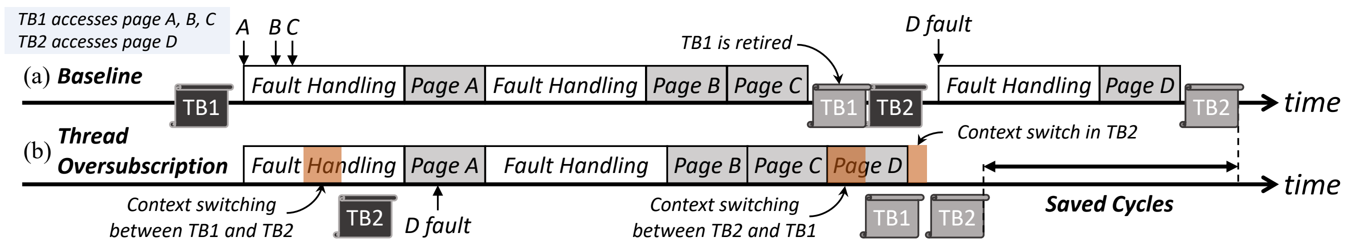

**Figure 7.** Overview of how thread oversubscription can increase the batch size. TB is short for thread block.

As we can see in the figure, thread oversubscription eliminates the need for the third batch, reducing the overall batch processing time. Note that the GPU runtime fault handling time for the second batch is slightly increased to handle an additional page fault for page D.

## 4.2 Unobtrusive Eviction

While the thread oversubscription technique alleviates the performance impact of demand paging by amortizing the GPU runtime fault handling time and reducing the number of batches, there is an opportunity to reduce the page migration time itself. Figure 8 shows the performance impact of page eviction by comparing the performance of a GPU with 50% memory oversubscription (denoted as baseline) to the performance of a GPU with unlimited memory, where no page evictions occur. The GPU with 50% memory oversubscription experiences an average performance loss of 46%. We compare this to the performance of a GPU with an instant page eviction capability (denoted as ideal eviction). Removing the page eviction latency achieves an average performance improvement of 16%. To this end, we propose unobtrusive eviction, a mechanism that eliminates the page eviction latency by overlapping page evictions with page migrations to the GPU.

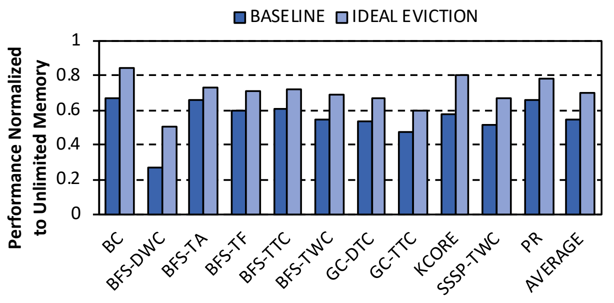

**Figure 8.** Performance of a GPU with 50% memory oversubscription compared to a GPU with unlimited memory, and how the performance changes with ideal eviction.

DMA engines in modern CPUs and GPUs [6, 7, 40, 44, 45] allow bidirectional transfers. However, page evictions and new page allocations are serialized to prevent the new page from overwriting the evicted page, as discussed in Section 3. Our goal is to devise a mechanism that exploits bidirectional transfers without violating the serialization requirement. The key idea is to *preemptively* initiate a single page's eviction and enable pipelined bidirectional transfers afterwards. To perform this preemptive eviction promptly at the beginning of batch processing, we modify the GPU runtime and add a new GPU memory status tracker. This does not affect the GPU runtime fault handling performance since the tracking is performed only when a new page is allocated in the GPU's memory.

When a page fault interrupt is raised by the GPU MMU, the top-half interrupt service routine (ISR) responds. It checks whether the number of GPU resident pages is at capacity via the GPU memory status tracker. If so, it sends a preemptive eviction request to the GPU. The rest of the fault handling (e.g., preprocessing of the page faults, CPU-side page table walks) is performed by the bottom-half ISR. Since the GPU runtime fault handling time (i.e., the time between when a batch's processing begins and when the first page migration for the batch begins) is at least tens of microseconds, the single page eviction is completed before the first page migration begins. By doing so, the first page migration can proceed without any delay. If a subsequent page eviction is required, the bottom-half ISR of the GPU runtime schedules the next page eviction along with the page migration to the GPU memory. Note that since the single page preemptive eviction is initiated by the GPU runtime when the batch processing begins, no additional overhead (e.g., communications with GPU) is required. Figure 9 shows how the unobtrusive eviction is implemented.

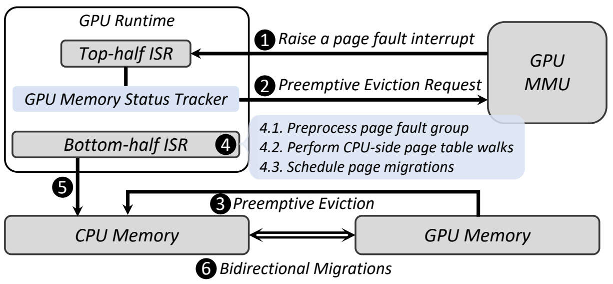
GPU Runtime
Top-half ISR
GPU Memory Status Tracker
Bottom-half ISR
4
1 Raise a page fault interrupt
2 Preemptive Eviction Request
GPU MMU
4.1. Preprocess page fault group
4.2. Perform CPU-side page table walks
4.3. Schedule page migrations
5
3 Preemptive Eviction
CPU Memory
GPU Memory
6 Bidirectional Migrations

**Figure 9.** Unobtrusive eviction scheme.

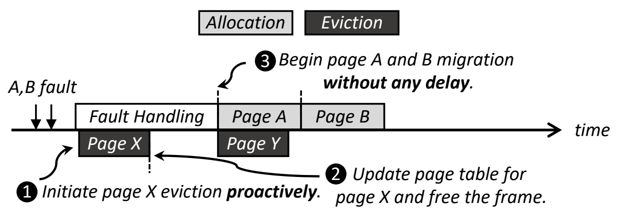
Allocation
Eviction
A,B fault
❸ Begin page A and B migration without any delay.
Fault Handling
Page A
Page B
time
Page X
Page Y
❶ Initiate page X eviction proactively.
❷ Update page table for page X and free the frame.

**Figure 10.** Overview of how unobtrusive eviction works.

Figure 10 provides a timeline example of how the unobtrusive eviction works. When the GPU runtime begins a batch's processing, it checks the GPU memory status. If it is at capacity, it initiates a single page eviction (❶). Once page X is evicted from the GPU's memory, both CPU and GPU page tables are updated (❷). Unlike the baseline case (Figure 4), page A can be migrated to the GPU memory without any delay (❸ ). At the same time, page Y can be evicted using bidirectional transfers. Since the data transfer speed from the GPU to CPU memory is faster than the other way around [29], eviction is completely unobtrusive and migrations to the GPU can occur without any delay.

## 5 Evaluation

In this section, we describe our evaluation methodology and evaluate our proposed techniques. We provide more detailed analyses in Section 6.

### 5.1 Methodology

***Simulator.*** We use MacSim [27], a cycle-level microarchitecture simulator. We modify the simulator to support virtual memory, demand paging [4, 5, 8, 9, 29, 38, 39, 45, 53], and the Virtual Thread (VT) [52]. Our virtual memory implementation includes TLBs, page tables, and a highly threaded page table walker [46, 47, 53]. Demand paging is modeled based on the GPU runtime software for NVIDIA PASCAL GPUs (driver v396.37) [4, 5, 38, 40, 41, 45]. We also model the batch processing mechanism that handles a multitude of outstanding faults together. We use a 1024-entry fault buffer [45] to handle up to 1024 simultaneous page faults. The page table walker is shared across all the SMs in the GPU, allowing up to 64 concurrent page walks [47]. Each TLB contains the miss-status-holding-registers (MSHRs) to track in-flight page table walks [53].

We use 20$\mu s$ as the conservative value for the GPU runtime fault handling time similar to other works [8, 9, 29, 44, 53]. The benefits of our mechanism would be more pronounced if this overhead were larger, which is often the case based on our profiling experiments. We also employ the state-of-the-art page prefetching mechanism [53]. The GPU memory capacity is configured to be fractions (50% and 75%) of the memory footprint of each workload as in prior works [29, 53]. We use LRU for page replacement policy [4, 5, 45]. We include the context switching overhead (i.e., timing overhead of storing and restoring context, such as register files and per-thread-block state information, to/from global memory whenever a context switch occurs) in the evaluation for thread oversubscription. For premature eviction rates, we calculate the running average of the lifetime of pages at every 100k cycles. The threshold is empirically set to 20% for our evaluation.

We also evaluate unobtrusive eviction using the simulator, in which we faithfully model the scheme described in Section 4.2. We were not able to implement the mechanism in real NVIDIA runtime software because the open source part of the runtime software does not include the part where it interacts with the driver entirely. Table 1 shows the configuration of the simulated system.

**Table 1.** Configuration of the simulated system.

| **GPU Configuration** | |
|---|---|
| **Core** | 16 SMs, 1GHz, 1024 threads per SM, 256KB register files per SM |
| **Private L1 Cache** | 16KB, 4-way, LRU, L1 misses are coalesced before accessing L2 |
| **Private L1 TLB** | 64 entries per core, fully associative, LRU |
| **Memory Configuration** | |
| **Shared L2 Cache** | 2MB total, 16-way, LRU |
| **Shared L2 TLB** | 1024 entries total, 32-way associative, LRU |
| **Memory** | 200 cycle latency |
| **Unified Memory Configuration** | |
| **Fault Buffer** | 1024 entries |
| **Fault Handling** | 64KB page size, 20$\mu s$ GPU runtime fault handling time, 15.75GB/s PCIe bandwidth |

***Workloads.*** We select 11 workloads from the GraphBIG benchmark suite [37]. These include Betweenness Centrality (BC), Breadth-First Search (BFS), Graph Coloring (GC), Single-Source Shortest Path (SSSP), K-core decomposition (KCORE), and Page Rank (PR). BC is an algorithm that detects the amount of influence a node has over the flow of information in a graph [33]. Graph traversal is the most fundamental operation of graph computing, for which we include five different implementations of BFS: data-warp-centric (DWC), topological-atomic (TA), topological-frontier (TF), topological-thread-centric (TTC), and topological-warp-centric (TWC). GC performs the assignment of labels or colors to the graph elements (i.e., vertices or edges) subject to certain constraints [22, 24], for which we include two different implementations: data-thread-centric (DTC) and

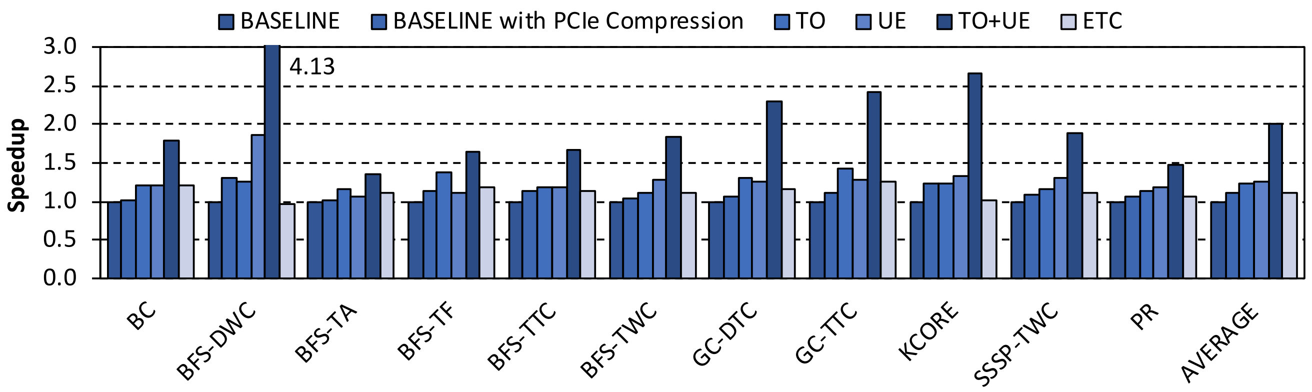

**Figure 11.** Performance comparison among baselines with the state-of-the-art page prefetching [53] with and without PCIe compression, eviction-throttling-compression (ETC) [29], and our proposed mechanisms (thread oversubscription is denoted as TO, and unobtrusive eviction is denoted as UE), normalized to the baseline.

topological-thread-centric (TTC). KCORE partitions a graph into layers from external to more central vertices [34]. SSSP finds the shortest path from the given source to each vertex, for which we include a topological-warp-centric (TWC) implementation. PR is an algorithm that evaluates the importance of web pages [16]. The footprints of these workloads range from 26MB to 349MB, with an average of 74MB. Impractically long simulation times prevent us from using the entire real-world data set. Note, however, that the footprints are larger than the ones used in prior works [29, 53].

## 5.2 Performance

Figure 11 shows the performance of the proposed thread oversubscription and unobtrusive eviction mechanisms (denoted as TO and UE, respectively). The performance is normalized to that of a baseline (denoted as BASELINE) that uses the state-of-the-art page prefetching technique, proposed by Zheng et al. [53]. We also evaluate how the performance changes when PCIe (de)compression is utilized (denoted as BASELINE with PCIe Compression). Last, we compare our mechanisms with the eviction-throttling-compression (ETC) mechanism [29]. Our mechanism (TO+UE) achieves an average speedup of 2x and 1.81x relative to the BASELINE and BASELINE with PCIe Compression, respectively. Our mechanism even outperforms ETC by 79% on average. ETC includes three components: proactive eviction (PE), memory-aware throttling (MT), and capacity compression (CC). However, the authors of ETC disable PE for irregular applications as it hurts performance, and we replicate this behavior.[7] The memory-aware throttling (MT) technique can be beneficial when memory is oversubscribed if it can reduce the working set size and thus decrease the page thrashing rates.[8] However, as we discussed in Section 1, since most of the memory pages are shared in many large-scale, irregular workloads, MT is not effective.

The performance improvement achieved by our mechanism is mainly attributed to the following two key factors. First, as described in Section 4.1, our thread oversubscription technique effectively reduces the total number of batches, thereby mitigating the overall page fault handling overhead. For the evaluated workloads, we see that the number of batches is reduced by 51% on average because we process 2.27x more page faults per batch, as shown in Figures 12 and 13. This corroborates our observation that it is beneficial to increase thread concurrency in the presence of frequent page migrations, even at the cost of expensive context switches.

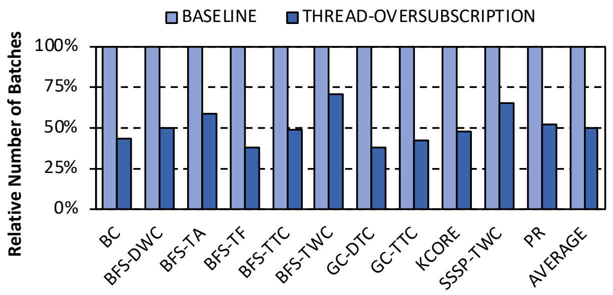

**Figure 12.** Total number of batches.

Second, the unobtrusive eviction technique reduces the average batch processing time significantly. As seen in Figure 14, when employed with the thread oversubscription technique, it reduces the average batch processing time by 60%. The unobtrusive eviction technique is more effective in larger batches, where the page migration time dominates the

[7]We faithfully model ETC to the best of our knowledge.

[8]When triggered, MT statically throttles half of the SMs in the beginning. After the initial phase, it repeats two epochs, the detection epoch and the execution epoch. Depending on the behavior monitored during the detection epoch, MT decides to throttle or unthrottle SMs.

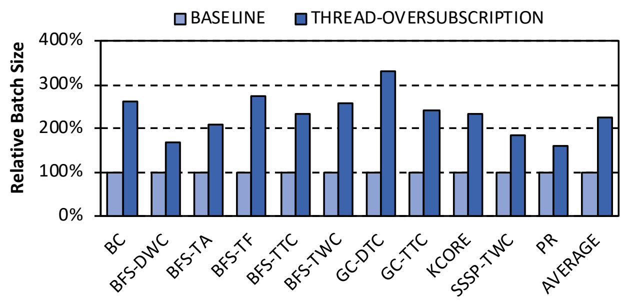

**Figure 13.** Average batch sizes.

overall batch processing time, and the thread oversubscription technique provides that opportunity. The unobtrusive eviction technique is particularly effective for BFS-DWC. From further investigation, we see that BFS-DWC has an extremely high divergent memory access pattern. Because of this, constant page thrashing occurs throughout the execution. Therefore, the unobtrusive eviction technique, which hides the eviction latency, leads to a 4.13x performance improvement. When both of them are employed (denoted as TO+UE), we find that the average batch processing time is reduced by 27% compared to the baseline even though we handle more page faults per batch. As a result, the thread oversubscription technique achieves 22% performance improvement and the unobtrusive eviction technique achieves 61% additional performance improvement, as shown in Figure 11.

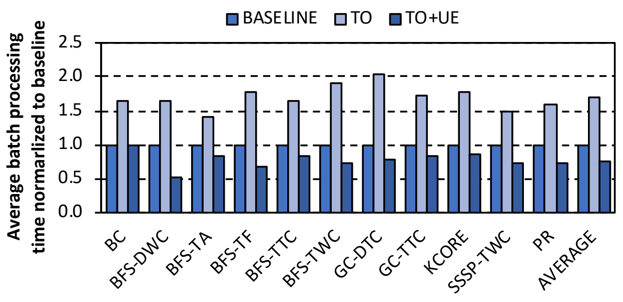

**Figure 14.** Average batch processing time.

## 6 Analysis

In this section, we provide in-depth analyses of our proposed mechanisms.

### 6.1 Effect on Premature Eviction

Since the thread oversubscription technique increases the batch size, premature evictions can also increase. Therefore, premature eviction is a key metric to evaluate the efficacy of the thread oversubscription technique. Figure 15 shows how the premature evictions change when the thread oversubscription technique is employed, compared to the baseline. Surprisingly, the thread oversubscription technique decreases premature evictions in most of the workloads. The reason is that it increases the likelihood of GPU resident pages being used before evicted. This is particularly the case for topological graph workloads since the input graph is traversed or processed in a topological order by each thread block in the topological graph workloads. Since the thread oversubscription technique increases thread concurrency and makes their memory accesses more parallel, there is a higher chance of concurrently running thread blocks accessing similar sets of pages while they are residing in the GPU memory.

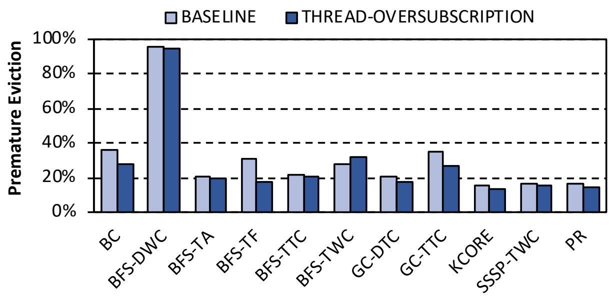

**Figure 15.** Premature eviction comparison.

The only exception is BFS-TWC. This happens when the working sets of existing thread blocks and additional thread blocks (context switched in) are distinct (as in the case of BFS-TWC), thrashing each other. We observe that premature evictions increase as we increase the number of concurrently runnable thread blocks for some workloads (e.g., BFS-TWC and SSSP-TWC). In the case of BFS-TWC, the premature evictions increase by 4% and 38% compared to the baseline as we increase the number of concurrently runnable thread blocks by 2x and 3x, respectively. In the case of SSSP-TWC, the premature evictions decrease by 2% but increased by 27% compared to the baseline as we increase the number of concurrently runnable thread blocks by 2x and 3x, respectively. As shown in Figure 15, however, this detrimental effect is delimited since our technique monitors the premature eviction rates and dynamically controls the degree of thread oversubscription, as described in Section 4.1.

### 6.2 Effect on Batch Size

Figure 16 compares the distribution of batch size. Batch size is computed as the summation of all the pages in each batch in size. Efficiency is computed as the reciprocal of an average time to handle each page. The line chart shows that as the number of page faults per batch increases, efficiency increases since the GPU runtime fault handling time is amortized. It is clearly seen that bigger batches appear when the

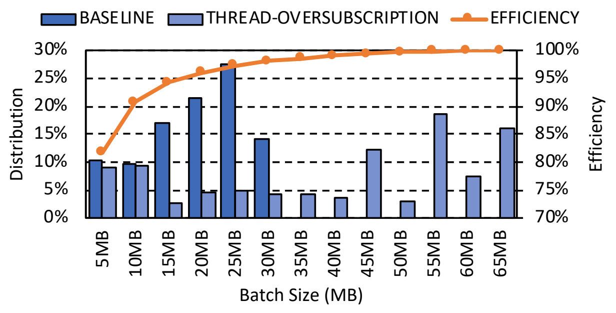

**Figure 16.** Batch size comparison.

thread oversubscription is employed. From this, we conclude that thread oversubscription effectively increases the batch size.

## 6.3 Sensitivity to Oversubscription Ratio

Figure 17 shows the performance impact of memory oversubscription (bar chart) and the performance improvement of the unobtrusive eviction technique (line chart) when the memory oversubscription ratio is varied from 0.1 (i.e., the GPU physical memory capacity is set to 10% of each application's working set size) to 1.0 (i.e., all application data fits in the GPU memory). We observe the following. First, the performance impact of memory oversubscription increases as the GPU memory becomes smaller. With smaller GPU memory, a smaller fraction of the application fits in the GPU memory. This causes more frequent page evictions to occur, requiring more page migrations between the CPU and GPU. Second, the unobtrusive eviction technique provides a scalable performance benefit. Obviously, when all application data fits in the GPU memory, it is ineffective (i.e., speedup of 1). As the memory size becomes smaller, its efficacy increases, as page evictions occur more frequently. The unobtrusive eviction technique provides an average speedup of 1.63x when the oversubscription ratio is 0.1. Therefore, we conclude that the unobtrusive eviction technique can provide a commensurate amount of performance benefit as the application requires more memory.

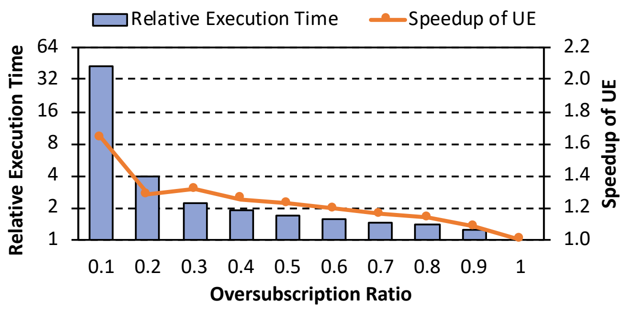

**Figure 17.** Sensitivity to memory oversubscription ratio.

## 6.4 Sensitivity to GPU Fault Handling Time

Figure 18 shows the average performance improvement of all workloads when the GPU runtime fault handling time is varied from 20$\mu$s to 50$\mu$s. Each bar is normalized to its own baseline (i.e., baseline configuration with 20$\mu$s, 30$\mu$s, 40$\mu$s, and 50$\mu$s). The data shows that the performance improvement that our proposed techniques can bring increases as the GPU runtime fault handling time becomes larger. To give perspective, we profile a regular application (e.g., `vectoradd`) in addition to the BFS measurement (Section 4.1). We observe that the GPU runtime fault handling time is higher for an irregular application (with a minimum of 50$\mu$s) than for a regular application (with a minimum of 20$\mu$s). Provided that a batch processing usually takes more than 300$\mu$s in real GPUs, we believe that our proposed techniques can bring higher performance improvement when employed in real GPUs.

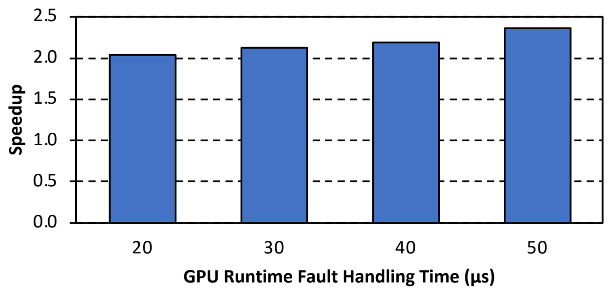

**Figure 18.** Sensitivity to the GPU fault handling time.

## 6.5 Context Switching Overhead

For thread oversubscription evaluation, we include the context switching overhead (i.e., timing overhead of storing and restoring context, such as register files and per-thread-block state information, to/from global memory whenever a context switching occurs). Although not shown, we also evaluated a close-to-ideal context switching overhead using an infinite-size shared memory. For this infinite-size shared memory, we used the latencies computed according to Equations 1 and 2 in [52] as follows:

$$Overhead\ (Cycles) = \frac{Context\ (Bits)}{Bandwidth\ (Bits\ Per\ Cycle)}$$

For example, assume that a shared memory has 32 banks, and each bank provides 32-bit data per cycle. The total bandwidth of the shared memory is 1024 bits per cycle. For a thread block that consists of 2048 threads where each thread uses 10 32-bit registers, the total context size is estimated to be 85KB (see footnote 5). In this case, the close-to-ideal context switching overhead is computed as 680 cycles, which is less than a microsecond. From this, we observed that the overall execution time is insensitive to the context switching overhead.

## 7 Related Work

To our knowledge, this paper provides the first comprehensive analysis of where the major inefficiencies arise in the page fault handling mechanism used in modern GPUs [4, 5, 38, 39, 45], and proposes a solution to mitigate the inefficiencies and regain the performance lost due to demand paging. This section briefly discusses related work.

***Virtual Memory Support in GPUs.*** Address translation is required for memory references in virtualized memory. The performance implications of address translation are widely known, and considerable research has been done to reduce the overheads [10–15, 25, 32, 35, 46, 47, 50, 51]. Power et al. [47] study a memory management unit (MMU) design for GPUs and propose per-compute unit TLBs and a shared page walk unit. Pichai et al. [46] also explore GPU MMUs and propose modest TLB and page table walker augmentations. Ausavarungnirun et al. [8] propose a GPU memory manager that provides an application-transparent multiple page size support in GPUs to increase the TLB reach. Ausavarungnirun et al. [9] propose an address-translation-aware GPU memory hierarchy design that reduces the overhead of address translation by prioritizing memory accesses for page table walks over data accesses. Shin et al. [48] propose a SIMT-aware page table walk scheduling mechanism to improve address translation performance in irregular GPU workloads. Cong et al. [17] propose a two-level TLB design for a unified virtual address space between the host CPU and customized accelerators.

***Demand Paging in GPUs.*** Unlike traditional GPUs, where GPU runtime must migrate all pages to the GPU memory before launching a kernel, modern GPUs support on-demand page faulting and migration (i.e., demand paging) [4, 5, 38, 39, 45]. The demand paging support eliminates the need for manual data migration, thereby reducing programmer effort and enabling GPU applications to compute across datasets that exceed the GPU memory capacity. However, its implication on performance is considerable and has been studied a lot recently [2, 3, 8, 9, 20, 26, 28, 29, 53]. Zheng et al. [53] explore the problem of PCIe bus being underutilized for demand-based page migration and propose a software page prefetcher to better utilize PCIe bus bandwidth and hide page migration overheads. Agarwal et al. [2] investigate the problem of demand-based page migration policy and propose using virtual-address-based program locality to enable aggressive prefetching and achieve bandwidth balancing. Agarwal et al. [3] further explore page placement policies and propose using characteristics (i.e., bandwidth) of heterogeneous memory systems and program-annotated hints to maximize GPU throughput on a heterogeneous memory system.

Li et al. [29] propose ETC, a memory management framework to improve GPU performance under memory oversubscription. The ETC framework categorizes applications into three categories (regular applications with and without data sharing, and irregular applications) and applies three techniques (proactive eviction, memory-aware throttling, and capacity compression) differently. The proactive eviction mechanism is ineffective (or rather detrimental) for irregular applications. The reason is that it heavily relies on predicting correct timing to avoid both early and late evictions. Since irregular applications access a large number of pages within a short period of time, it becomes ineffective. However, our UE mechanism relies on neither the timing nor size prediction. All we need is to initiate a single page eviction before the first page migration begins. This is based on our in-depth analysis of how GPU runtime schedules page migrations and how it interacts with the DMA engine and the GPU hardware (via command buffer). Their memory-aware throttling and capacity compression techniques are orthogonal to our work.

## 8 Conclusion

We proposed a GPU runtime software and hardware solution that enables efficient demand paging for GPUs. To this end, we first inspected a real GPU runtime software implementation to find major sources of inefficiencies in the page fault handling mechanisms employed in modern GPUs. Based on two key observations we made (the large scale serialization introduced by the batch processing mechanism and the serialization observed in the page migrations), we proposed two co-operative techniques: (1) thread oversubscription, which increases the batch size to amortize the GPU runtime fault handling time and reduce the number of batches, and (2) unobtrusive eviction, which eliminates the page migration delay caused by page evictions. Our extensive evaluation showed that the proposed solution achieves an average speedup of 2x over the baseline that employs the state-of-the-art prefetching.

## Acknowledgments

We thank the anonymous reviewers and HPArch group members for their feedback. This research was partially supported by Intel Corporation.

## References

[1] Advanced Micro Devices Inc. 2012. AMD Graphics Cores Next (GCN) Architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[2] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W. Keckler, and Thomas F. Wenisch. 2015. Unlocking Bandwidth for GPUs in CC-NUMA Systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA).*

[3] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[4] Nikoly Akhenykh. 2017. Unified Memory On Pascal and Volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf.

[5] Nikoly Akhenykh. 2018. Everything You Need to Know About Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf.

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

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

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[9] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[11] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[12] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[13] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[14] Abhishek Bhattacharjee and Margaret Martonosi. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[15] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-core Cooperative TLB for Chip Multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[16] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the International Conference on World Wide Web (WWW)*.

[17] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[18] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. A Survey on Thread-Level Speculation Techniques. *ACM Comput. Surv.* 49, 2, Article 22 (June 2016), 39 pages. https://doi.org/10.1145/2938369

[19] Xu Fan, Shen Li, and Wang Zhiying. 2012. HVD-TLS: A Novel Framework of Thread Level Speculation. In *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*.

[20] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay Between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[21] Google. [n.d.]. Google GPUs Cloud Computing. https://cloud.google.com/gpu.

[22] Andre Vincent Pascal Grosset, Peihong Zhu, Shusen Liu, Suresh Venkatasubramanian, and Mary Hall. 2011. Evaluating Graph Coloring on GPUs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[23] IBM. [n.d.]. GPUs Cloud Computing. https://www.ibm.com/cloud/gpu.

[24] Mark T. Jones and Paul E. Plassmann. 1993. A Parallel Graph Coloring Heuristic. *SIAM J. Sci. Comput.* 14, 3 (May 1993), 654–669. http://dx.doi.org/10.1137/0914041

[25] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[26] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*.

[27] Hyesoon Kim, Jaekyu Lee, Nagesh B. Lakshminarayana, Jaewoong Sim, Jieun Lim, Tri Pho, Hyojong Kim, and Ramyad Hadidi. 2012. *MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide.*

[28] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2014. VAST: The illusion of a large memory space for GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[29] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[30] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[31] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. 2010. Speculative Execution on GPU: An Exploratory Study. In *International Conference on Parallel Processing (ICPP)*.

[32] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.* 10, 1, Article 2 (April 2013), 38 pages. https://doi.org/10.1145/2445572.2445574

[33] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda. 2009. A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets. In *Proceedings of the International Conference on Parallel and Distributed Processing (IPDPS)*.

[34] David W. Matula and Leland L. Beck. 1983. Smallest-last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (July 1983), 417–427. http://doi.acm.org/10.1145/2402.322385

[35] Collin McCurdy, Alan L. Coxa, and Jeffrey Vetter. 2008. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[36] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[37] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the

Context of Industrial Solutions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[38] NVIDIA Corp. 2016. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[39] NVIDIA Corp. 2016. NVIDIA Tesla V100. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[40] NVIDIA Corp. 2017. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/index.html.

[41] NVIDIA Corp. 2018. NVIDIA Driver Downloads. https://www.nvidia.com.

[42] NVIDIA Corp. 2019. NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[43] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[44] PCI-SIG. 2015. PCI Express Base Specification Revision 3.1a.

[45] Peng Wang. 2017. UNIFIED MEMORY ON P100. https://www.olcf.ornl.gov/wp-content/uploads/2018/02/SummitDev_Unified-Memory.pdf.

[46] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[47] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[48] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[49] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling Preemptive Multiprogramming on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[50] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. 2018. Filtering Translation Bandwidth with Virtual Caching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[51] Hongil Yoon and Gurindar S. Sohi. 2016. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[52] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. 2016. Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[53] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.