Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory

TYLER ALLEN, University of North Carolina at Charlotte, USA
BENNETT COOPER and RONG GE, Clemson University, USA

The abstraction of a shared memory space over separate CPU and GPU memory domains has eased the burden of portability for many HPC codebases. However, users pay for ease of use provided by system-managed memory with a moderate-to-high performance overhead. NVIDIA Unified Virtual Memory (UVM) is currently the primary real-world implementation of such abstraction and offers a functionally equivalent testbed for in-depth performance study for both UVM and future Linux Heterogeneous Memory Management (HMM) compatible systems. The continued advocacy for UVM and HMM motivates improvement of the underlying system. We focus on UVM-based systems and investigate the root causes of UVM overhead, a non-trivial task due to complex interactions of multiple hardware and software constituents and the desired cost granularity.

In our prior work, we delved deeply into UVM system architecture and showed internal behaviors of page fault servicing in batches. We provided quantitative evaluation of batch handling for various applications under different scenarios, including prefetching and oversubscription. We revealed that the driver workload depends on the interactions among application access patterns, GPU hardware constraints, and host OS components. Host OS components have significant overhead present across implementations, warranting close attention.

This extension furthers our prior study in three aspects: fine-grain cost analysis and breakdown, extension to multiple GPUs, and investigation of platforms with different GPU-GPU interconnects. We take a top-down approach to quantitative batch analysis and uncover how constituent component costs accumulate and overlap, governed by synchronous and asynchronous operations. Our multi-GPU analysis shows reduced cost of GPU-GPU batch workloads compared to CPU-GPU workloads. We further demonstrate that while specialized interconnects, NVLink, can improve batch cost, their benefits are limited by host OS software overhead and GPU oversubscription. This study serves as a proxy for future shared memory systems, such as those that interface with HMM, and the development of interconnects.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; **Processors and memory architectures**; • **Software and its engineering → Virtual memory**; **Distributed memory**; • **Hardware → Hardware accelerators;**

Extension of Conference Paper [4]: This extension makes the following additional contributions: (1) quantitatively analyzes the batch cost and isolates the task components, and classifies them based on synchronous and asynchronous UVM operations; (2) investigates UVM costs in multi-GPU computing and reveals how the host OS operations change with involved GPU devices; and (3) examines multiple platforms with different generations of GPUs and interconnects. These extensions bring new insights into the dominant costs and root causes and the potential benefits and limitations of advanced hardware features and platforms.

This work is supported in part by the U.S. National Science Foundation under Grants CNS-CCF-1942182.

Authors' addresses: T. Allen, University of North Carolina at Charlotte, 210E Woodward Hall, Charlotte, NC, 28223-0001; e-mail: t.allen@uncc.edu; B. Cooper and R. Ge, Clemson University, 100 McAdams Hall, Clemson, SC, 29634-0901; e-mails: {bwc, rge}@clemson.edu.

1544-3566/2024/01-ART14 $15.00
https://doi.org/10.1145/3632953

Additional Key Words and Phrases: Unified Virtual Memory, Heterogeneous Memory Management, Virtual Memory, GPGPU, Accelerated Computing

ACM Reference format:
Tyler Allen, Bennett Cooper, and Rong Ge. 2024. Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory . *ACM Trans. Arch. Code Optim.* 21, 1, Article 14 (January 2024), 24 pages.
https://doi.org/10.1145/3632953

1 INTRODUCTION

Graphics Processing Units (GPUs) have become a computational mainstay in modern HPC systems and paved the way for other accelerators in the HPC space. Discrete GPUs have separate physical memory traditionally programmed through APIs and managed by device drivers. Multiple technologies are being developed to abstract the complexity of separate CPU and GPU physical memory domains to ease the burden of programming and increase codebase portability. ***Heterogeneous Memory Management*** **(HMM)** and NVIDIA **Unified Virtual Memory (UVM)** are two independent yet potentially collaborative efforts. These technologies integrate device memory domains into the OS virtual memory system and transparently migrate pages on demand across devices. HMM is a Linux kernel feature that provides a generic interface for heterogeneous memory management to vendor- and device-specific drivers on commodity systems [15, 26]. NVIDIA UVM presently offers an all-in-one approach combining paging and device drivers for NVIDIA GPUs. It can also integrate with the HMM interface [44]. As of today, NVIDIA UVM has been prolific, adopted by the US Department of Energy and in common HPC frameworks such as Raja [9], Kokkos [11], and Trilinos [25].

Transparent demand paging and migration come with heavy performance costs, as noted by prior studies [3, 23, 27–29, 52]. Figure 1 shows that the access latency generally increases by one or more orders of magnitude compared to explicit direct management by programmers. While such costs may be acceptable for applications with in-core computing on GPU memory, high-performance systems suffer inefficient utilization as a consequence. Further, the out-of-core or oversubscription capability comes at a much greater cost, largely prohibitive for most applications. Prefetching mitigates but cannot overcome all of the costs, and could even increase it for some memory-oversubscribed workloads [3, 19, 21, 28, 50, 51].

Understanding the overhead sources in transparent paging and migration is essential, especially as the cost of delegating management to the OS through HMM will be imposed on any system using the HMM interface. HMM may become the de facto technology with the ongoing advocates and development efforts. However, HMM is not yet well supported on commodity systems. In this work, we focus on NVIDIA UVM technology. As we reason in Section 2, UVM offers a functionally equivalent testbed for a novel low-level performance study for both UVM and future HMM-compatible systems. Using UVM, we can identify the root sources of performance concerns and attribute them to their roles in HMM-based implementations.

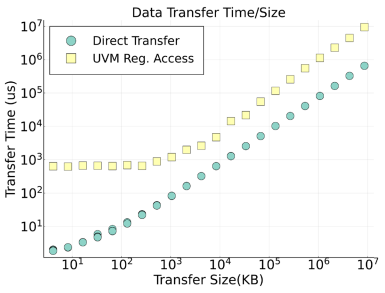


Fig. 1. Access latency with unified virtual memory increases by one or more orders of magnitude over explicit direct management.

In our prior work [4], we take a deep dive into the UVM system architecture and the internal behaviors of page fault generation and servicing. We perform extensive analyses on the UVM

driver workload's basic units: page fault *batches* or groups of GPU-generated page faults. We instrument the `nvidia-uvm` driver to collect metadata containing targeted high-resolution timers and counters for specific batch events, routines, and page fault arrival. Through extensive experimentation and quantitative analyses, we obtain insights into where the UVM costs originate and where performance optimization or design reconsiderations are applicable for UVM, HMM, and future vendor-specific HMM systems.

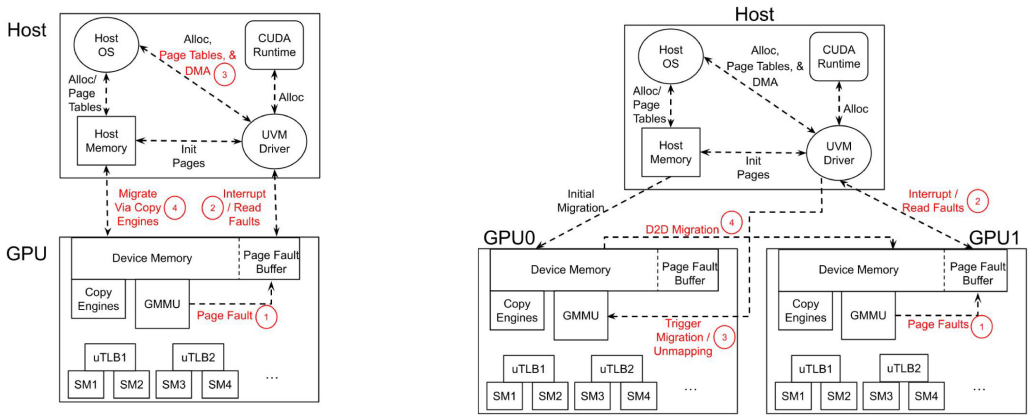
This article includes significant extensions to our prior work [4]. First, we take a top-down approach to quantitative cost analysis and isolation and uncover how the constituent costs accumulate and overlap, governed by synchronous and asynchronous operations. It locates the root sources of dominant costs and potential gains from hardware and software improvements. Second, we extend our study from single GPU to multi-GPU computing and show how batch workloads are distributed among GPUs and how host OS operations change. Third, we investigate multiple platforms with different CPU-GPU and GPU-GPU interconnects including PCIe 3.0 and NVLink2. We highlight the common costs shared by all platforms and pinpoint the components that benefit interconnect hardware and software improvement.

Overall, this article examines a great breadth of functionality in UVM in a fine granularity over multiple systems and provides a more complete and realistic understanding of UVM across different devices and interconnects. We make the following main contributions:

- We conduct an in-depth study of the UVM system fault generation, batching and servicing, and the core UVM work unit, offering perspective and rationale behind the design decisions and constraints of the UVM system architecture.
- We quantitatively analyze the batch cost, its distribution over devices and software components, and the impacts of prefetching and GPU memory oversubscription. We further isolate the component costs based on synchronous and asynchronous UVM operations, pinpointing dominant costs and root causes.
- Using UVM as an example of future HMM systems, we show that the driver workload depends on the interactions among application access patterns, GPU hardware constraints, and host OS components. We further isolate performance considerations to vendor-specific and common codes among all implementations and discuss improvements for different cases.
- We extend to multi-GPU computing, revealing how the host OS operations change with involved GPU devices.
- We examine multiple platforms with different generations of GPUs and CPU-GPU/GPU-GPU interconnects, demonstrating the potential benefits and limitations specialized hardware can have on the UVM system.

2 UVM BACKGROUND AND RELATED WORK

NVIDIA UVM provides Linux-like virtual memory between multiple computing devices through paging, where *page faults* trigger *data migration* between host memory and accelerators. This is aligned with the functional philosophy of HMM, a Linux kernel API that allows heterogeneous device drivers to have first-class access to Linux page tables [15, 26]. The goal of UVM and HMM is aligned, and the introduction of HMM will extend the functionality of UVM and UVM-like systems to allow a unified memory management interface for user applications. **The NVIDIA UVM driver is among the first backend solutions to interface with HMM. However, to the best of our knowledge, the full integration for x86_64 systems is yet under development [30, 44].** Provided that NVIDIA UVM is currently the primary real-world implementation of transparent paging and migration across memory domains, we focus on UVM but draw insights applicable to HMM.



(a) Single GPU System (b) Multi-GPU System

Fig. 2. The client-server UVM architecture. The clients are user-level GPU or host code, and the server is the UVM driver residing on the host. Illustration of fault initiation and servicing between the host and a single GPU (a) and between the host and two GPUs (b). Numbers are a forward reference to Figure 3 and are discussed in detail in Section 4.

In this section, we provide an overview of the UVM system architecture and functionality. Also, we note where these systems intersect and overlap with HMM support.

2.1 The UVM Architecture

UVM uses a client-server architecture between one or more software clients (user-level GPU or host code) and the server (host driver) servicing page faults for all clients. The UVM host driver on the host is open source with dependencies on the proprietary `nvidia` driver/resource manager and the host OS for memory management. This driver is a runtime fault servicing engine and the memory manager for managed memory allocations. Figure 2 illustrates the architecture with one GPU client (a) and two GPU clients (b).

Any active thread on the GPU can trigger a page fault, as shown in Figure 2. Similarly, a thread on the host can also trigger a page fault. Take a GPU page fault on a single GPU platform (Figure 2(a)) for example: the fault is generated and handled by the hardware thread's corresponding µTLB [39] ①. The thread may continue executing instructions not blocked by a memory dependency. Meanwhile, the fault propagates to the **GPU memory management unit (GMMU)**, which writes the corresponding fault information into the *GPU Fault Buffer* and sends a hardware interrupt to the host ②. The fault buffer acts as a circular array, configured and managed by the UVM driver [39]. The `nvidia-uvm` driver fetches the fault information, caches it on the host, and services the faults through page processing ③ and page migration ④, where the former involves page table update and TLB shootdown on the host and GPU page table update, and the latter involves page migration.

The GPU exposes two functionalities to the host via the GPU command push-buffer—host-to-GPU memory copy and fault replay. As part of the fault servicing process, the driver instructs the GPU to copy pages into its memory, generally using high-performance hardware "copy engines." Once the GPU's page tables are updated and the data is successfully migrated, the driver issues a fault replay [53], which clears the waiting status of µTLB, causing them to "replay" the prior miss.

On a multiple GPU system, page processing ③ and page migration ④ may not be fully performed on the host anymore. This change results in reduced UVM overhead. Take the scenario in Figure 2(b) for example: GPU1 threads incur page faults and demand data residing on GPU0. Page processing that involves page unmapping and TLB shootdown now occurs on GPU0, resulting

in less cost on the host due to offloaded work and overlapped GPU TLB shootdown and page table updates [3]. Meanwhile, page migration is from GPU0 to GPU1 and benefits from high-speed GPU-GPU interconnects such as NVLink. More details about the cost reduction are presented in Sections 4 and 6.

2.2 Fault Batching

GPU page fault delivery to the host requires two steps. First, the GPU sends an interrupt over the interconnect to alert the host UVM driver of a page fault. The interrupt wakes up a worker thread to begin fault servicing if none is awake. Second, the host retrieves the complete fault information from the *GPU Fault Buffer*.

The `nvidia-uvm` driver groups outstanding faults into **batches** in the host-side cache. The **fault batch** is the fundamental unit of work. Batching allows the driver to ignore most interrupts and thus serves as an optimization. The default fault retrieval policy reads faults until the batch size limit (i.e., 256 faults) is reached or no faults remain in the buffer. The worker thread services one batch after another and sleeps if it finds no new faults. Fault batching and fault handling policies are the UVM driver's independent decisions. For reference, device drivers are still responsible for these actions in HMM implementations.

Pages in the same fault batch are processed and migrated in the same time step. For compatibility with the host OS and future HMM implementations, UVM adopts the host OS's page size for migration and tracking: 4KB pages for x86_64 systems and 64KB pages for Power9 systems. UVM has additional internal abstraction for management and performance considerations. For x86_64, pages are upgraded from 4KB to 64KB within the UVM runtime *as a component of prefetching*, emulating the 64KB Power9 page size. Additionally, the driver splits all memory allocations into 2MB logical **Virtual Address Blocks (VABlocks)** on GPUs. These VABlocks serve as logical boundaries; the driver processes all batch faults within a single VABlock together, and each VABlock within a batch requires a distinct processing step. UVM also tracks all physical GPU memory allocations from the `nvidia` resource manager and evicts allocations at the VABlock granularity when needed.

2.3 Related Work

Prior work is primarily in three categories: (1) high-level analysis of UVM at the application level and attempts in optimizing UVM performance for specific applications or problem spaces, (2) alterations to hardware or migration of software functionality into hardware via simulation, and (3) lower-level analysis of UVM functionality in system software. Prior works do not perform deep, fine-grain quantitative cost analysis on existing systems and architectures in the same level of detail we present.

High-level Analysis and Application Optimization. High-level analysis typically focuses on either comparing UVM to traditional manually managed memory applications or comparing UVM across different hardware platforms such as Power9 vs. x86 and NVLINK vs. PCIe. The overall performance impact of UVM was studied in [28, 29, 52] on several applications for both non-oversubscription and oversubscription. Manian et al. study UVM performance and its cooperation with MPI across several MPI implementations [35]. Gu et al. produce a suite of benchmarks based on the Rodinia benchmark suite to perform these kinds of evaluations [23]. Markidis et al. focus on advanced features of UVM, such as runtime allocation hints and properties [14], while Gayatri et al. focus on the impacts of prefetching and Power9 **Address Translation Services (ATSs)** [21]. Several works have tried to improve graph-processing or graph-specific applications that have known irregular processing by utilizing the remote mapping (DMA) capabilities of UVM as well as altering access patterns or data ordering to make accesses less irregular [22, 36, 38]. Shao et al. provide application-level runtime analysis for oversubscribed multi-GPU workloads under UVM [45].

Table 1. Experimental Platforms with Two Generations of GPUs and Two Types of GPU-GPU Interconnects

Platform	Host			NVidia GPU			Interconnect	
	CPU	OS	MEM	GPU (count)	UVM Driver	MEM	CPU-GPU	GPU-GPU
I	32-core Epyc	Fedora 33	128GB	Titan V (1)	460.27.04	12GB HBM2	PCIe 3.0	NA
II	Dual 20-core Xeon	CentOS 8.2.2004	372GB	Tesla V100 (2)	470.42.01	16GB HMB2	PCIe 3.0	PCIe 3.0
III	Dual 20-core Xeon	CentOS 8.2.2004	372GB	Tesla V100 (2)	470.42.01	16GB HMB2	PCIe 3.0	NVLink2

Hardware and System Alterations. Some works discuss fundamental changes to the UVM architecture or UVM hardware to improve overall performance, whereas our work focuses on identifying performance characteristics and issues that are solvable on existing hardware/software. Griffin offers architectural changes to enhance page locality for multi-GPU systems [7]. Kim et al. simulate "virtual threads" to effectively increase the overall number of threads resident on the GPU to better hide latency, along with increasing the fault batch size to allow the host to process more faults at the same time [27]. Several works suggest replacements for UVM that diverge from the demand-paging paradigm [6, 37]. Ganguly et al. use the existing but sparsely utilized page counters system within the existing UVM ecosystem to improve performance for memory-oversubscribed workloads [20], offer modifications to eviction and prefetching algorithms after integrating these features into hardware [19], and present a runtime framework that uses online CPU-GPU interconnect traffic to dynamically choose a suitable memory management strategy in oversubscribed workloads [18]. Similarly, Yu et al. also offer architectural changes to coordinate eviction and prefetching [50]. Chang et al. instrument the UVM driver with an adaptive page migration scheme to better serve temporal locality of GPU caches and spatial locality of GPU memory [12]. Li et al. propose a dynamic memory management framework to curb the cost of oversubscription in UVM by incorporating proactive eviction, memory-aware throttling, and capacity compression [32]. Li et al. further analyze the page walk of GPU local page faults in multi-GPU workloads and offer a mechanism of short-circuiting the page walk process to improve performance when migrating data between GPUs [31].

Other works discuss similar system alterations outside the context of UVM and propose alternative management schema. Haria et al. propose using a memory paradigm between physical addressing and virtual addressing for system accelerators [24]. Qureshi et al. make a case for allowing GPUs to manage accesses into **Non-volatile Memory express (NVMe)** backing storages to reduce the overhead of CPU-centric storage access [42]. Suchy et al. incorporate their compiler and runtime-based address translation into the Linux kernel as an alternative to the paging mechanism used in virtual memory to reduce the overhead introduced by paging [47].

UVM System Analysis. These works are the most similar to ours. Allen and Ge focus on the driver-level performance of prefetching, showing page-level access patterns and performance data for the general case, but not the root source of UVM costs [3]. Kim et al. show an example of batch-level size/performance data similar to ours [27]. In contrast, our work dives into the software- and hardware-based root causes under different scenarios and quantitatively analyzes the constructions of batches and cost breakdown.

3 EXPERIMENTAL ENVIRONMENT

We conduct experiments on three platforms presented in Table 1, one with a single GPU (Platform I) and two with two GPUs (Platforms II and III). There are two generations of NVidia GPUs (Titan V vs. Tesla V100) and two types of GPU-GPU interconnects (PCIe 3.0 vs. NVLink2).

We collect all data through a modified UVM driver distributed alongside the NVIDIA driver. We modify the UVM driver into two versions. One logs per-fault metadata for gathering overall

statistics about faults such as their GPU SM of origin. The other is instrumented with targeted high-precision timers and event counters for collecting batch-level data. Counters and timers are used to collect higher-level events and timestamps in order to minimize the overhead. Batch data is logged to the system log at the end of each batch using a reliable custom tool.

We use the applications in Table 2 for case studies. They are representative HPC applications; i.e., the kernels include `sgemm`, Gauss-Seidel, and FFT and are commonly used in various HPC applications, and HPGMG is a full proxy application representing algebraic multigrid methods.

Table 2. Benchmarks Used in Evaluation and Analysis

Benchmark	HPC Use Examples
cuBLAS sgemm	Fluid Dynamics [46], Finite Element [8], Deep Learning [13]
stream	Memory bandwidth (triad-only) [16]
cuFFT	LAMMPs [40, 48], Particle Apps [41], Molecular Dynamics [48], Deep Learning [33]
Gauss-Seidel	HPCG [17], AMR [10]
HPGMG-FV	Proxy App for AMR [1]

4 OVERALL COST AND BREAKDOWN FOR BASE CASES

As discussed, a fault batch is the unit of UVM driver workload. Here we delve into the UVM driver to find out what task components are involved in batch handling and the associated costs. We concentrate on the **base cases without prefetching or oversubscription**. These base cases serve as the references for evaluating prefetching and oversubscription in later sections. We further compare and contrast page migrations in three system settings with varying source devices and interconnects: CPU-GPU over PCIe, GPU-GPU over PCIe, and GPU-GPU over NVLink.

4.1 Timeline of Tasks in Batch Handling

CPU-GPU Page Migration. The UVM driver takes a batch as the basic unit and processes one batch after another. Figure 3 shows the timeline of an individual batch for the CPU-GPU page migration. Each batch consists of a sequence of tasks in order. Some tasks are completed by the host alone, while two tasks, i.e., service faults and replay, need the collaboration of the destination GPU. For these two tasks, the host requests the GPU to asynchronously perform its parts and waits for the completion using a barrier at the end of the batch.

In this case, the host performs the following task components:

- **Fetch Faults**: Fetch entries from the GPU fault buffer, decode them, and store them in the batch context.
- **Preprocess**: Deduplicate the faults in the batch; sort them by VA space, fault address, and access type; and generate an ordered view of the faults.
- **Service Faults**: Scan through the ordered view of faults, group them by different VA blocks, and service the groups. Servicing involves the operating system for kernel work, such as page unmapping and TLB shootdown. The host also enqueues asynchronous tasks to the GPU to complete servicing.
- **Push Replay**: Request the GPU to replay faults asynchronously.
- **Flush Fault Buffer**: Request the GPU to flush the fault buffer, preparing for the next iteration of faults.
- **Tracker Wait**: Wait for all outstanding asynchronous GPU work to be completed before completing the current batch. This serves as a synchronization barrier.

The GPU performs two asynchronous tasks to complete fault service and fault replay, respectively. These tasks overlap with CPU tasks and thus their time costs are (partially) hidden.

- **Service Faults**: Fetch pages from the host using DMA, and map page and update pagetable to make page information consistent between the host and GPU.
- **Replay Fault**: GPU MMU issues fault replay requests to uTLBs, which replay data access.

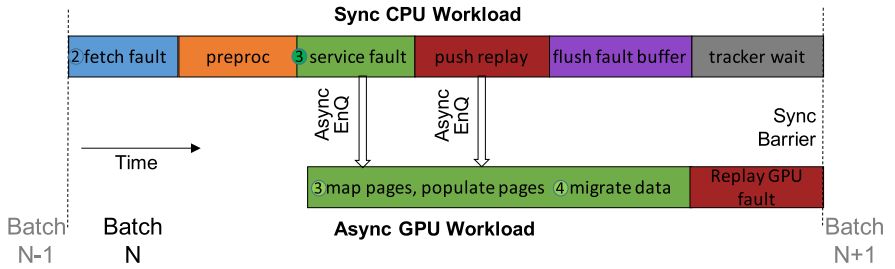


Fig. 3. Timeline of major tasks for a single UVM batch for the CPU-GPU page migration. The host driver performs a sequence of tasks in order while enqueueing two asynchronous tasks to the GPU device. The host and the device synchronize to complete the current batch using a barrier. The circled numbers correspond to those in Figure 2.

GPU-GPU Page Migration. In this case (not shown in Figure 3), the pages originally reside at the source GPU instead of the host. The task of fault servicing is different from that in the CPU-GPU case, while other tasks are the same. Specifically, the host initiates fault servicing and triggers the source GPU to asynchronously unmap pages and shoot down TLB entries. The destination GPU has the same asynchronous workload as shown in Figure 3 but migrates data from the source GPU rather than the host.

4.2 Batch Cost and Breakdown

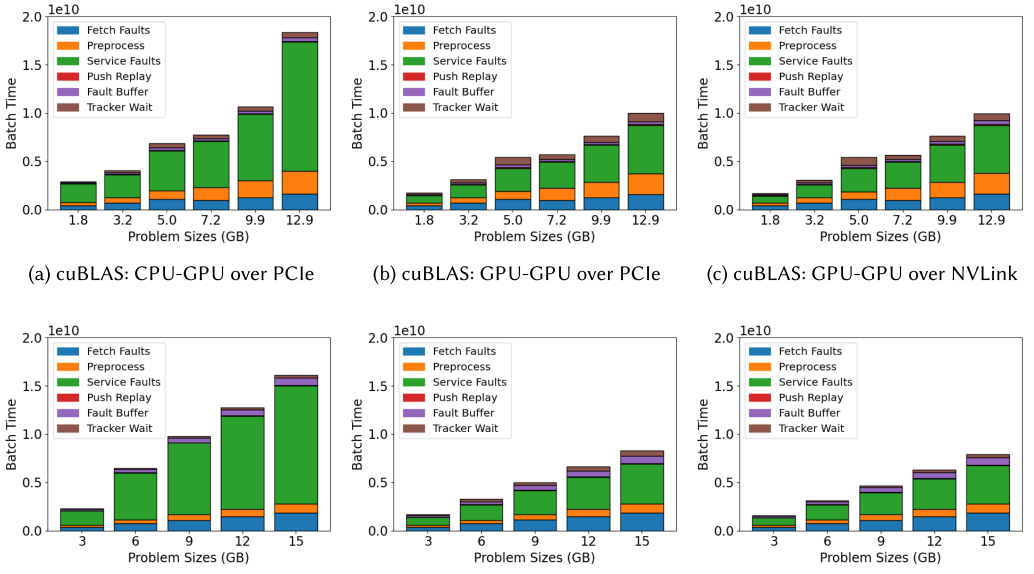
Now we examine the time cost for batch handling. We obtain the timing by instrumenting the top-level function of the UVM driver that handles each individual batch and the task functions it further invokes. While such timing is performed on the host, it captures the global cost and encapsulates the asynchronous GPU tasks through the track wait time. We study three system settings: CPU-GPU over PCIe on Platform I, GPU-GPU over PCIe on Platform II, and GPU-GPU over NVLink on Platform III.

CPU-GPU On-demand Page Migration. Figure 4(a) shows the overall cost of batch handling and the breakdown by the task components for cuBLAS sgemm on Platform I. The GPU generates page faults and migrates pages on demand that reside on the CPU. As expected, the overall batch time increases with the matrix size, and so do the component times. This is explained by the fact that larger problems have more data and a larger number of pages, resulting in more page faults and on-demand page migration.

Among the task components, fault servicing dominates, accounting for over 70% of the total time. This cost is attributed to necessary actions performed by the operating system kernel. Such actions include unmapping pages from the host pagetable and shooting down the TLB entries so that the pages can be migrated to the GPU and mapped to the GPU pagetable. TLB shootdown is costly on multicore systems because it involves the TLBs on each core. In addition, locks are frequently used to prevent race conditions and coherency.

Fault fetching and preprocessing incur non-negligible costs, accounting for about 12% of the total cost, respectively. Both task components are synchronous. The former involves fault metadata transfer over the CPU-GPU interconnect, which has large access latency. The latter traverses the fault cache and sorts the faults.

For cuBLAS sgemm on Platform I, the tracker wait time is relatively small, suggesting that the asynchronous GPU tasks are completed in a similar amount of time as the CPU tasks. That is, the costs of GPU pagetable update and page migration are effectively overlapped. Two other task components, fault replay and buffer flush, incur negligible costs.



(a) cuBLAS: CPU-GPU over PCIe (b) cuBLAS: GPU-GPU over PCIe (c) cuBLAS: GPU-GPU over NVLink

(d) Stream: CPU-GPU over PCIe (e) Stream: GPU-GPU over PCIe (f) Stream: GPU-GPU over NVLink

Fig. 4. Distribution of time in nanoseconds (ns) among the UVM driver components across all batches.

GPU-GPU On-demand Page Migration. Figure 4(b) shows the experimental results of GPU-GPU on-demand page migration for cuBLAS sgemm on Platform II. The GPU-GPU case is similar to CPU-GPU in several aspects. First, the total batch time increases with the problem size. Second, servicing time dominates and increases with the problem size. Third, fault fetching and preprocessing do not vary with system settings, indicating the same host tasks for both cases.

We note several differences by contrasting the two cases. First, the total time in Figure 4(b) is significantly smaller, i.e., by 60%. This reduction is mainly attributed to servicing time. Second, while still dominating, servicing time only accounts for about 50% of the total time. Third, while still insignificant, the tracker wait time is doubled.

The stark decrease in fault servicing time is due to the actual page table updates offloaded to the source GPU. Fault servicing includes operations such as page unmapping and TLB shootdown on the source device. In the CPU-GPU case, the host is actively involved and responsible for all these required page table updates. In addition, TLB shootdown is particularly costly for synchronization operations on multi-core CPU architectures with the software-based coherency [2, 5]. In the GPU-GPU case, the host does not need to update its own page tables but mainly orchestrates work on the GPUs. GPU page table updates and TLB shootdown are hardware based and relatively much faster [49]. As a result, the host fault servicing time is more than halved.

The tracker wait time increase is noteworthy, even though it does not significantly contribute to the total batch time. Tracker wait time is the synchronization barrier between the host driver and the GPUs and measures the time the host waits for outstanding asynchronous GPU tasks to complete. The increase indicates that the asynchronous GPU tasks are less overlapped than for the CPU-GPU case. Tracker wait time would be more significant when advanced features of UVM are enabled. These advanced features include prefetching and oversubscription; their impact is discussed in Section 6.

The sum of service time and tracker wait time is halved in the GPU-GPU case. The reduction indicates two pieces of information: (1) the overall improvement due to page table updates offloaded

from the host to GPU and (2) the more efficient memory allocation and page handling by the GPU than the host, which complies with the findings in [2].

Figure 4(c) shows the cost of GPU-GPU demand paging over NVLink for cuBLAS `sgemm` on Platform III. Contrary to the intuition that GPU-GPU over NVLink could improve time over GPU-GPU over PCIe, the total cost and components do not vary with the interconnect for basic on-demand paging without prefetching and oversubscription. This observation suggests that basic on-demand paging is unable to sufficiently use the bandwidth of either PCIe or NVLink.

4.3 Variations among Applications

Here we examine if the observations hold across the applications. Specifically, these observations are (1) the dominance of the service time across platforms, (2) the decrease of service time of GPU-GPU, and (3) the increase of tracker wait time of GPU-GPU. We note that the full service time is the sum of service time and tracker wait time, provided the replay time is relatively much smaller.

Figures 4(d) through 4(f) show the batch time for Stream. All observations maintain, but with a smaller increase in the tracker wait time. Given that the main asynchronous component is the actual migration of data, we attribute the smaller increase in tracker wait time to Stream's linear access pattern, which allows for greater coalescing of data. These common observations motivate the in-depth investigation of the impacting factors of the service time, which is discussed in the following section.

5 ATTRIBUTES OF THE UVM DRIVER WORKLOAD

To understand the cause of the service time and its dominance, we investigate several key workload features and quantitatively analyze them. This section presents the details of the following findings.

- Data movement: Data transfer from the source to the destination can incur a high cost but, counterintuitively, is not the dominating factor. In addition, its cost can be fully or partially overlapped due to asynchronicity.
- Host OS interaction: Some components, such as CPU page unmapping, require the host OS and incur surprisingly significant overhead on the fault path.
- Fault distribution/access pattern: The distribution of faults over 2MB VABlocks determines the trend for performance variance. The patterns are inherent to applications.

We first study the CPU-GPU setting on Platform I and then discuss the difference between the GPU-GPU settings on Platforms II and III.

5.1 In-depth Examination of the CPU-GPU Case

5.1.1 Data Movement. Data movement is the primary purpose of the UVM driver and sets the trend for performance. Data amount is the leading performance indicator in most UVM scenarios for a given batch. Figure 5 demonstrates that the average batch cost rises linearly with the amount of data moved for all applications. However, the average cost differs with applications, and there is a high variance for a given application.

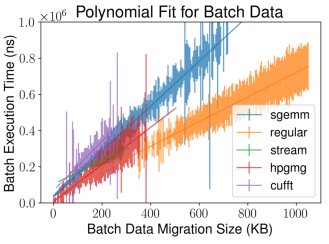


Fig. 5. Best fit of batch time vs. batch data amount migrated for one run of several applications.

Even though data movement is the primary purpose, its direct cost is not primary in a fault batch. Instead, management is far more costly.

We use the example of a moderately sized `sgemm` to demonstrate this point. Figure 6 shows that transfer time accounts for less than 25% of the total batch time for almost all batches. This observation offers two insights: (1) Most batch servicing time is *not* spent on data transfer. While faster hardware may benefit performance, the more significant issue is ensuring the driver efficiently utilizes the interconnect subsystem. (2) The variance and skew must be derived from batch characteristics, the driver software, and the driver's interaction with the host OS and hardware. We investigate the constituent components of the overall performance cost, including variance, in the remainder of this section.

5.1.2 Host OS Interaction. Management operations for host memory frequently require expensive interactions with the host OS. The host component of UVM is built on top of the existing virtual memory system in the Linux kernel. Because of this, migrations are subject to additional latencies incurred by existing mappings and the underlying virtual memory subsystem. We use an existing, UVM-optimized application to demonstrate this issue—the HPGMG implementation provided by NVIDIA [43].

Figure 7 shows an example of CPU-side behavior influencing GPU fault performance outcomes. The two subfigures show the same problem with the same configuration, except (a) uses a single OpenMP thread, whereas (b) uses the default OpenMP thread configuration (one thread per logical core). Notably, the former configuration shows roughly twice the performance by simply disabling multithreading, and the performance trend falls in line with other applications that we have seen for a given data size.

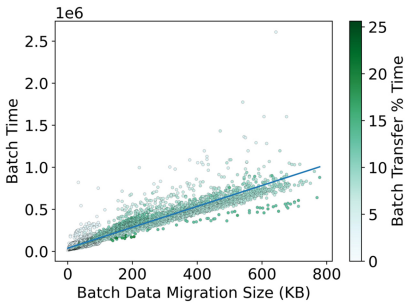


Fig. 6. The percentage of time spent per batch performing data transfer for `sgemm`. At most, the transfer time is approximately 25% of the total batch time but is typically far lower.

Further, page unmapping represents a significant portion of execution time for many batches, as represented by the tone of color in Figure 7. Page unmapping is an operation in the existing virtual memory system on the host that UVM extends to support faults from GPU. Page unmapping is performed when the GPU touches a VABlock that is partially resident on the CPU. In this scenario, the driver calls into the kernel function `unmap_mapping_range()` to unmap all pages within the VABlock residing in host memory as part of the page migration. Interestingly, we observe that OpenMP multithreading exaggerates this specific cost for HPGMG. We note that this behavior does not occur in trivial cases, such as parallelizing data initialization in the `sgemm` application, indicating that data access patterns and thread affinity play a role in this issue.

We draw two conclusions about host OS interaction from the data presented: (1) unmapping host-side data takes place on the fault path and incurs significant overhead, and (2) certain host-side parallelizations of an application using UVM can exaggerate these unmapping costs. The host OS performs this operation, and the costs likely stem from issues with virtual mappings across CPU cores, flushing dirty pages from caches and TLBs, NUMA, and other memory-adjacent issues. Additionally, these operations do not take place in bulk due to the logical separation of VABlocks within UVM. This is an area that deserves particular scrutiny as HMM also performs host page unmapping on the fault path using host OS mechanisms, implying a similar cost could be applied to all devices when using HMM [15, 26]. Design and implementation issues such as how unmapping takes place and if it needs to be performed on demand deserve further investigation.

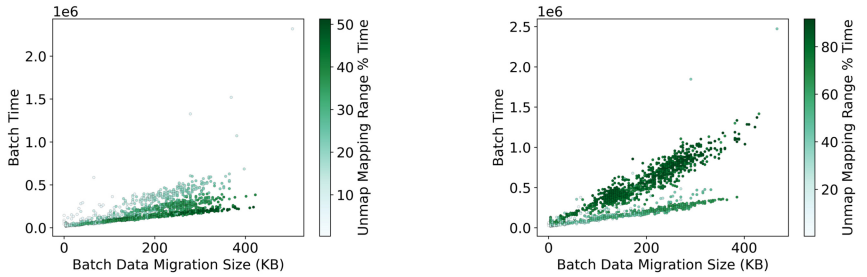


Fig. 7. Single-threaded HPGMG (a) and multithreaded HPGMG (b), where the percentage indicates relative time per batch spent unmapping host-resident pages. Multithreading incurs larger percentages for unmapping.

5.1.3 Fault Distribution/Access Patterns. We next examine the distribution of faults in a batch in the memory space, e.g., spatial locality at the page granularity. Within the UVM driver, all operations are logically separated on VABlock (page-aligned 2MB) regions, making VABlocks a source of performance variation. Figure 8 shows batches colored by the number of unique VABlocks present in the data transfers for each batch. While each batch is subject to other sources of variance, one major trend is that, for batches with similar workloads, more VABlocks incur higher costs and cause more significant performance variation. This behavior is consistent with our earlier observation that the driver handles VABlocks within a batch independently.

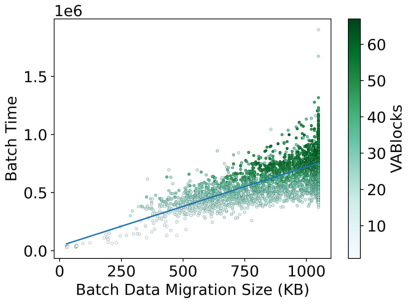


Fig. 8. Effects of VABlocks in batches on the batch time. For the same data migration size, a higher cost is associated with more VABlocks.

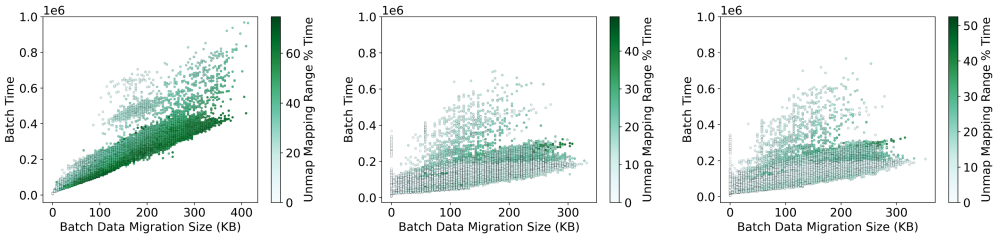
Processing each VABlock in parallel would be an intuitive optimization based on the driver design but would be highly workload imbalanced due to the large standard deviation in per-batch VABlock representation. In Table 3, there is a wide variation in the number of VABlocks present in each batch, and these distributions change with application. Additionally, there is a high variance in the number of faults per VABlock. As discussed in our prior work [4], the root cause of this inconsistency is that each fault batch contains pages from almost every SM on the GPU. Batched faults originate from different execution contexts, with only a few pages representing each SM. The sole benchmark with low variance is random access as it consistently has no locality within a single VABlock but still represents a very small workload per VABlock.

Table 3. VABlock Source Statistics in a Batch

	VABlock /Batch	Faults /VABlock	Std. Dev.	Min.	Max.
Regular	41.27	5.93	5.10	1	83
Random	233.09	1.04	0.20	1	6
sgemm	6.96	9.81	16.58	1	128
stream	3.93	15.37	8.17	1	72
cufft	25.14	2.89	2.22	1	129
gauss-seidel	2.31	22.44	27.96	1	208
hpgmg	2.39	13.62	15.72	1	212

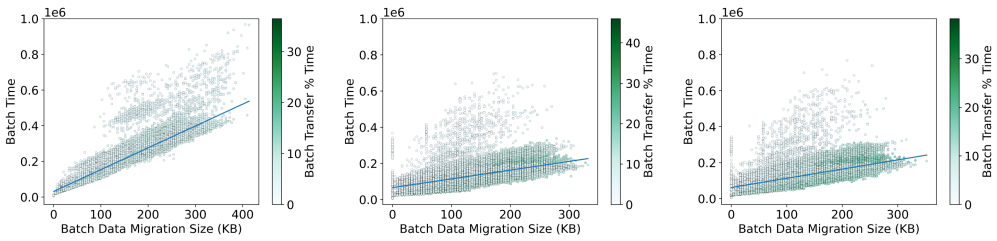
5.2 Improvement in the GPU-GPU Cases

As indicated by Figure 4, the host incurs a smaller cost in servicing faults in the GPU-GPU case than in the CPU-GPU case. We attribute the reduction to unmapping/unmapping offloaded to the source GPU and possibly



(a) CPU-GPU over PCIe (b) GPU-GPU over PCIe (c) GPU-GPU over NVLink

Fig. 9. The weights of page unmapping on the host for sgemm on three platforms. In the GPU-GPU cases, the host incurs much less time by mainly orchestrating the unmapping on the source GPU and mapping on the destination GPU.



(a) CPU-GPU over PCIe (b) GPU-GPU over PCIe (c) GPU-GPU over NVLink

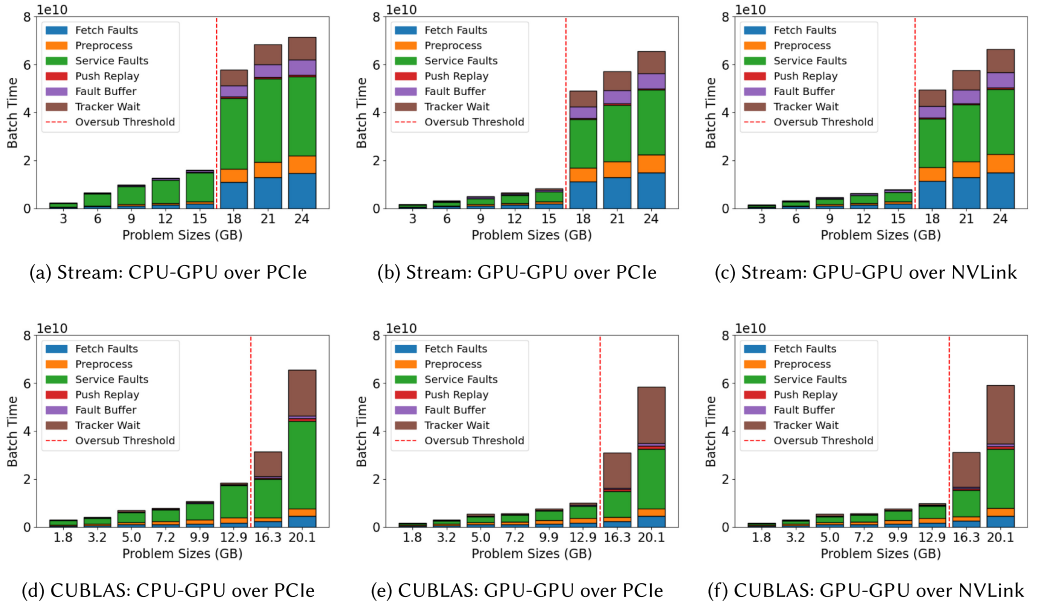
Fig. 10. The weights of data transfer setup time on the host for sgemm on three platforms. In the GPU-GPU cases, the host incurs much less time by mainly orchestrating the unmapping on the source GPU and mapping on the destination GPU.

less transfer time. In this subsection, we examine these explanations using experimental results and confirm their correctness.

Figure 9 presents the weights of page unmapping and TLB shootdown on the host batch time on three platforms. The host spends 50% on average of batching time in page unmapping and page mapping, which is encapsulated in serving faults in the CPU-GPU case on Platform I, in comparison to 15% in the GPU-GPU cases. This is because in the latter cases, the host mainly orchestrates the activities on the two GPUs without updating its own page tables. Figure 10 presents the weights of data transfer on the host batch time on three platforms. The data transfer has a similar percentage of the total batch time across the platforms. The higher total time in the CPU-GPU case indicates a larger absolute data transfer time.

6 EFFECTS OF PREFETCHING AND OVERSUBSCRIPTION

In practice, UVM offers two features by default to support its use in real applications—prefetching and oversubscription. Prefetching is fundamental to allowing UVM applications to achieve performance comparable to programmer-managed memory applications [3]. Oversubscription further simplifies programming, allowing applications to work with out-of-core data, but typically at a high performance cost. In this section, we analyze these two features, primarily identifying (1) how costs from the prior section translate into real workloads, (2) how prefetching and oversubscription impact batches qualitatively and quantitatively, and (3) how they differ with system settings.



(a) Stream: CPU-GPU over PCIe (b) Stream: GPU-GPU over PCIe (c) Stream: GPU-GPU over NVLink

(d) CUBLAS: CPU-GPU over PCIe (e) CUBLAS: GPU-GPU over PCIe (f) CUBLAS: GPU-GPU over NVLink

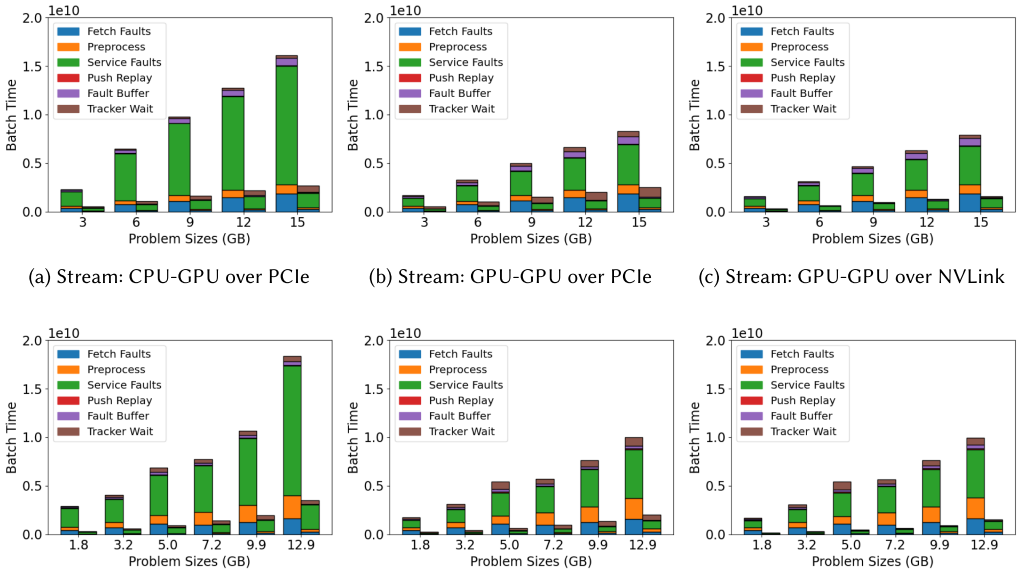
Fig. 11. Overall batch time and breakdown for oversubscribed applications. The red vertical dashed lines delimit non-oversubscription and oversubscription. Top: Stream. Bottom: CUBLAS. Left: CPU-GPU over PCIe. Center: GPU-GPU over PCIe. Right: GPU-GPU over NVLink.

6.1 Overall Performance Impacts

6.1.1 Oversubscription. Oversubscription causes one major change to the driver workload: once destination GPU memory is filled, the UVM driver must evict some VABlocks before migrating the on-demand pages. Eviction presents an immediate detriment in performance because each eviction is associated with page table updates on both the source and the destination devices and data transfer between them. Eviction is handled solely in the service task component.

Figure 11 shows the performance of the UVM driver for Stream and `sgemm`. Once GPU memory is oversubscribed, batch time increases by more than 3X. Essentially, every batch task has a much higher cost. Oversubscription also causes a shift in the distribution of batch components. Service time is still a dominating factor. Tracker wait time grows to a substantial portion of the batch time (e.g,. for `sgemm`), indicating that eviction makes the GPU's workload greater than the CPU's workload. By inspecting the source code of UVM, we find that eviction involves a three-step process: try to allocate memory space on the GPU but with a failure, evict victim VABlocks, and re-try to allocate space. While each step incurs overhead, the eviction is especially costly, as detailed in Section 6.2.1. Furthermore, the host memory is the hard-coded destination for eviction. This causes GPU-GPU migration to span three memory locations: host memory, the source GPU's memory, and the destination GPU's memory.

The advantage of GPU-GPU disappears for oversubscribed problems, causing the total times across the three platforms to be similar for both benchmarks. This total time is determined by eviction, which is the same among the three platforms. The increase in tracker wait time offsets the decrease in service time using GPU-GPU. In contrast to the significant performance gain for non-oversubscribed problems, GPU-GPU only achieves 1.17X speedup over CPU-GPU for oversubscribed sgemm, and GPU-GPU over NVLink additionally achieves a 1.05X speedup.



(a) Stream: CPU-GPU over PCIe (b) Stream: GPU-GPU over PCIe (c) Stream: GPU-GPU over NVLink

(d) CUBLAS: CPU-GPU over PCIe (e) CUBLAS: GPU-GPU over PCIe (f) CUBLAS: GPU-GPU over NVLink

Fig. 12. Overall batch time and breakdown for prefetching enabled workloads. Each plot shows a side-by-side comparison of non-prefetching (left bars) and prefetching (right bars).

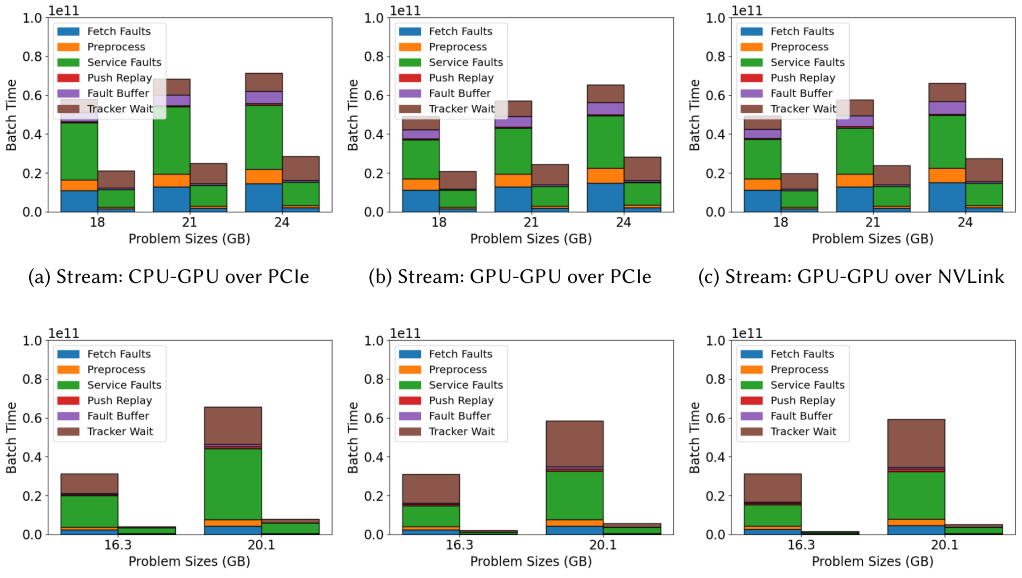
6.1.2 Prefetching. UVM utilizes a runtime prefetching routine as part of the default behavior. Upon a page fault, UVM fetches not only the requested page but also other pages in the same VABlock. Through prefetching, UVM amortizes the transfer latency and obtains a high bandwidth over the interconnect. Prefetching reduces the total number of faults generated during application executions and increases the coverage of pages in a batch. Like eviction, prefetching is handled entirely in the service component of the driver.

Compared to the base case, prefetching boosts the performance of the UVM driver by roughly an order of magnitude as shown in Figure 12. Except for tracker wait time, all other tasks' times are smaller with prefetching. This is mainly due to the reduced number of faults and batches: the prefetched pages satisfy the requests in the future and avoid page faults. The same application generates fewer faults and batches.

Tracker wait time with prefetching increases significantly and can be on par with service time. This increase indicates that the host waits longer for the asynchronous GPU tasks to complete the batches. Prefetching migrates a larger amount of pages in each batch compared to the base case. These additional pages require data transfer over the interconnect and page table updates on the GPU, leading to increased costs that are harder for the host tasks to overlap.

With prefetching, GPU-GPU over PCIe has a decreased advantage relative to CPU-GPU over PCIe, i.e., ~1.09X vs. 2.0X without prefetching. In addition, GPU-GPU over NVLink outperforms GPU-GPU over PCIe. It obtains a much higher speed, e.g.,., 1.7X vs. 1.09X for Stream. Specifically, the higher bandwidth and lower latency of NVLink reduce the data transfer time offloaded to the GPU. This in turn allows for better overlap of the synchronous and asynchronous workloads, which is manifested through the smaller track wait time.

6.1.3 Oversubscription + Prefetching. Coupling these two features results in more interesting behavior and aligns with real-world usage. In oversubscribed problems, prefetching may migrate pages that will be evicted before use. The costly evictions diminish the benefits of prefetching.



(a) Stream: CPU-GPU over PCIe (b) Stream: GPU-GPU over PCIe (c) Stream: GPU-GPU over NVLink

(d) CUBLAS: CPU-GPU over PCIe (e) CUBLAS: GPU-GPU over PCIe (f) CUBLAS: GPU-GPU over NVLink

Fig. 13. Overall batch time and breakdown for oversubscribed, prefetching-enabled workloads. Each plot shows a side-by-side comparison of non-prefetching (left bars) and prefetching (right bars).

Prefetching still boosts performance for some oversubscribed problems, but the gain depends on the application. Stream and `sgemm` have significantly different results for prefetching-enabled oversubscribed problems, as shown in Figure 13. Prefetching boosts the performance of Stream by 2.5X for oversubscribed problems over the executions without prefetching. This is a reduction from the 10X speedup for non-oversubscribed problems. Further, the advantage of GPU-GPU and NVLink disappear, and all three hardware configurations have a similar total time.

Prefetching still greatly boosts the performance of oversubscribed `sgemm` problems, i.e., by 8X compared to executions without prefetching. It is a reduction from the 10X speedup for non-oversubscribed problems. The advantage of GPU-GPU and NVlink still maintains for `sgemm`.

6.2 Detailed Analysis of the CPU-GPU Case

6.2.1 Oversubscription. Oversubscription allows applications to exceed GPU memory capacity by using a form of LRU eviction to swap pages back to the host. Upon a page fault, when GPU memory is oversubscribed, the UVM driver automatically evicts "cold" data back to the host to make room for new data at the granularity of 2MB VABlock. During this process, the GPU device must first identify a victim VABlock, unmap it in the pagetable, and transfer the block to the host.

The eviction incurs significant cost. Figure 14(a) shows batch timing data for `Stream` using a problem size that exceeds GPU memory. The batch distribution follows a somewhat expected trend: many batches are executed before full GPU memory allocation without requiring eviction, and others (colored) evict one or more VABlocks. Predictably, blocks containing evictions incur greater overheads to (1) fail allocation, (2) evict a VABlock and migrate the data back to the host, and (3) restart the block migration process, including host unmapping, data transfer, GPU mapping, and *page population*, a process by which pages are filled with zero values before data is migrated to them.

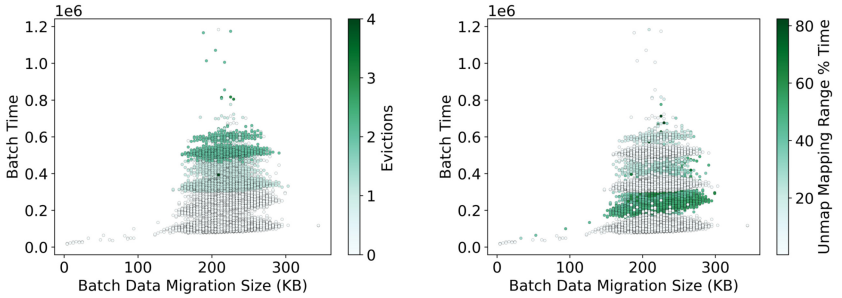


Fig. 14. Stream under oversubscription. Left: Multiple "levels" for the same eviction count. Right: A level may not include a portion of CPU unmapping.

In addition to the overhead of eviction, the batch time under oversubscription also depends on whether the VABlock on demand has been previously evicted from the GPU and remapped to the host. If the VABlock on demand is resident on the CPU and mapped on the host pagetable, then both costs of GPU eviction and CPU-GPU migration are accounted for in the overall time. In contrast, if a VABlock has been evicted from GPU but not remapped to the host, then the host does not have to re-pay the large `unmap_mapping_ranges()` cost for migration back to the device, cutting a significant portion of the time and creating the lower-cost levels of batches. This property is seen by comparing the pair of figures in Figure 14, where the lower "level" for the same number of evictions always has near-zero unmapping range cost.

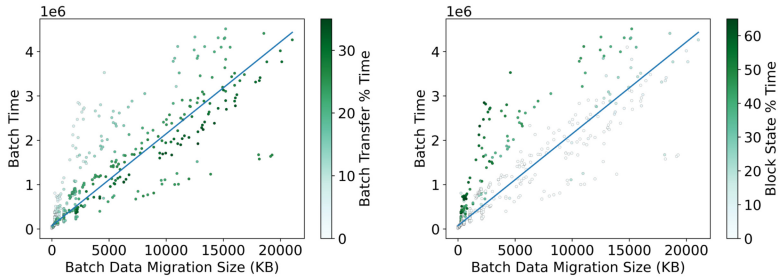
6.2.2 Prefetching. The prefetching mechanism is a type of *density prefetching*, sometimes called *tree-based prefetching*, and is described in detail in [3, 19, 27]. The prefetcher's scope is limited to a single VABlock and is only reactive; the prefetcher only flags pages within a VABlock currently being serviced for faults up to the full VABlock.

Prefetching reduces the number of batches (data points) by 93%; compare Figure 15(a) against Figure 6 for the previously seen `sgemm` with prefetching disabled. The relative performance trend is similar to the non-prefetching trend. However, some batches have highly exaggerated sizes due to large prefetching regions, i.e., 20MB vs. 800KB.

Many instances of very high-cost batches would have been considered outliers. These batches are traceable to the behavior seen in Figure 15(b), showing that up to 64% of batch time is spent in GPU VABlock state initialization not present in other batches. This time is largely attributed to two operations: (1) create DMA mappings for every page in the VABlock to the GPU, so that the GPU can copy data between the host and GPU within that region, and (2) create reverse DMA address mappings and store them in a radix tree data structure implemented in the mainline Linux kernel. These mappings are compulsory when a VABlock is first accessed and cannot be eliminated by prefetching. However, not every batch requires these DMA mappings to have the same high cost. In-line timing during these high-cost DMA batches shows that the majority of time is spent in the radix tree portion of this operation, indicating some performance issues potentially associated with that data structure. We do not present the data here for the low-level timing creates significant skew in the overall timing information.

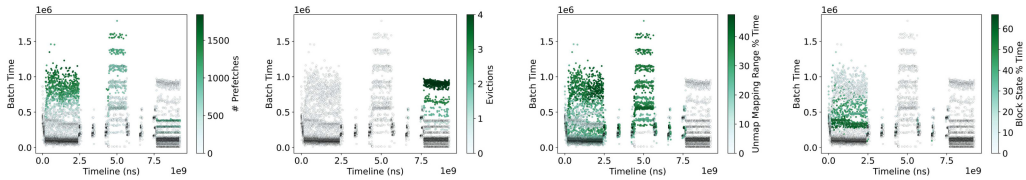
The overall characteristic of prefetching shows that reducing the overall number of batches is highly effective in speeding up UVM, even though it means performing larger quantities of work in the short term. The side effect is that prefetching makes the inconsistent DMA mapping take up a more significant proportion of the overall cost.

6.2.3 Eviction + Prefetching. Finally, eviction combined with prefetching creates the most complex scenario. Prior work has shown that the combination of prefetching and eviction can harm



(a) Reduced Number of Batches (b) Outliers Caused by DMA Mapping

Fig. 15. Batch profiles of sgemm with prefetching enabled. The mid-range cost batches are significantly reduced, and the high-end outliers correspond to negative performance impacts from creating and storing DMA mappings.



(a) #prefetches (b) #evictions (c) %(un)mapping (d) %VABlock DMA setup

Fig. 16. Batch profiles of sgemm as a time series. Prefetching occurs throughout the execution and evictions typically occur later. Unmapping and GPU state setup occur regularly throughout the application, and GPU state setup does not always have excessively high overhead.

performance for applications with irregular access patterns [3, 27, 52]. We evaluate this scenario by comparing prefetching-enabled and -disabled scenarios for the same applications.

Figure 16 shows sgemm with combined eviction and prefetching properties as a time series. While not shown, the range of data transfers is still extended, but not to the full 20MB range observed in the prefetching example alone; we attribute this to reduced block access density for the larger problem size.

We confirm that prefetching is still active and driving the larger batch sizes. Prefetching tends to happen earlier where the VABlock is consistently resident on the CPU, and subsequent accesses to the same VABlock can drive a robust prefetching response. Eviction ranges are remarkably similar to the non-prefetching data-set, fitting into the same sizes and ranges. The eviction set has relatively low batch sizes because evictions are caused by paging in *new* VABlocks, which have low access density at first. Non-eviction batches that include new VABlocks tend to have smaller batch sizes but must pay the high CPU unmapping cost discussed in the prior section. CPU unmapping cost can occur at any time during execution as new VABlocks are touched but tend to diminish later in execution after each VABlock has been touched by the GPU at least once. Though it is intermittent, creating DMA mappings can still have a high overhead. The high overhead may be caused by the growth of the underlying radix tree, but further investigation is required.

Overall, we confirm our intuition about *when* these batch features may occur and confirm that many of the cost relationships discussed earlier still account for a large quantity of runtime even with eviction and prefetching enabled. Additionally, we find that eviction costs need to be

optimized independently from the host OS performance. Prefetching can significantly reduce the total number of batches but is unable to mitigate the cost of each remaining batch.

6.3 Benefits of NVLink

Platform III has the same configuration as Platform II, except that it has the NVLink2 GPU-GPU interconnect vs. the PCIe GPU-GPU interconnect. Compared to PCIe 3.0, NVLink2 has higher peak bandwidth (i.e., 5X), lower latency (i.e., 45%), and cache coherence [34]. Here we discuss how NVLink's advantage is leveraged in UVM.

First, the base cases fail to leverage the benefits of NVLink. Without prefetching, only the on-demand pages are transferred across the interconnect. These pages are insufficient to saturate either PCIe or NVLink bandwidth. Meanwhile, all page faults are handled in the critical path for which the host requests the GPUs to service the faults and transfer the pages. The synchronous cost on the host is the bottleneck, determining the total cost.

Second, prefetching leverages the benefits of NVLink by transferring a large amount of data in each batch and reducing the total number of batches and page faults. Meanwhile, the host incurs less time due to reduced page faults. The higher peak bandwidth of NVLink results in a less asynchronous cost of larger batches that appear with prefetching. Consequently, NVLink does not suffer the same growth of tracker wait time in the case of prefetching non-subscribed problem sizes. In contrast, the larger batches saturate PCIe bandwidth, causing the host to wait for synchronization.

Third, oversubscription diminishes the benefits of NVLink2. Oversubscription, as it is currently implemented, always evicts pages back to the host memory. This causes the CPU-GPU PCIe interconnect to become active for data eviction. Depending on the amount of oversubscription and the applications, the CPU-GPU PCIe interconnect can quickly become the bottleneck, causing the increase in tracker wait time and limiting the overall performance. We anticipate that the advantage of GPU-GPU over NVLink2 maintains if the CPU-GPU interconnect was NVLink2.

7 DISCUSSION

7.1 Performance Factors

Data movement unexpectedly contributes only a small percentage of the overall cost. This suggests that improvements to basic hardware, such as interconnect bandwidth and latency, would still improve performance but would not eliminate the underlying bottlenecks.

Host OS overhead. Page table updates, particularly page mapping and unmapping, are the dominating factor. Some user code parallelization schemes can exacerbate these costs. Page table updates are three-party tasks, including the host UVM driver making requests and the migration source and destination devices asynchronously performing updates. In the case where the host serves as the migration source or destination, page (un)mapping and DMA setup are particularly costly, as they take place on the fault path and these operations are slower on CPUs than GPUs. Page (un)mapping is not intended to happen in frequent bursts with real-time constraints, as is the case with UVM and HMM. With the projection that HMM will be common code to support various devices, and UVM is commonplace today, further investigation is necessary to determine if page (un)mapping functionality can be improved to (1) incur less overall overhead and (2) avoid excessive costs based on the chosen parallelization of user applications. Alternatively, performing these operations asynchronously and preemptively may be preferable for GPU computing.

Driver Serialization. The driver, which orchestrates page table updates and page migration among the source and destination, is serial. Theoretically, the driver could handle the VABlocks in parallel. However, our workload analysis shows that parallelization would create a severe

workload imbalance. Parallelizing faults per SM may be more reasonable if devices support targeted per-SM replay. While these workload features are specific to NVIDIA GPUs, any vendor implementing HMM for parallel devices will encounter similar concerns and delays.

Oversubscription and eviction provide additional programmer flexibility. Oversubscription significantly increases the costs due to the required eviction from the GPUs. Even though the host OS does not intervene, the combined OS and eviction costs are exceedingly high. More cost-effective oversubscription requires optimization independent of the host OS as the underlying performance issues stem from the eviction algorithms and user applications.

Prefetching is an effective performance optimization technology by eliminating large numbers of fault batches. However, prefetching cannot mitigate the cost of individual batches with high DMA and CPU unmapping overhead. The current prefetching strategy is constrained within a VABlock and synchronous on the critical path. Increasing the prefetching scope to more than one VABlock or asynchronous prefetching could be more beneficial to regular workloads but may also complicate eviction. Prefetching and eviction must be co-developed for devices for optimal performance.

7.2 Multi-GPU Computing

Three-memory Command and Data Path. Multi-GPU computing essentially makes the UVM command and data path include three entities: host, source GPU, and destination GPU. The host UVM driver sends requests to the source and destination GPUs and orchestrates their operations. The data path consists of segments for page migration from the source GPU memory to the destination and segments for page eviction from the destination GPU memory to the host. GPU-GPU demand paging is faster when no eviction is involved, thanks to the faster page table updates and TLB shootdown by GPUs.

Oversubscription and Eviction. Eviction causes the host memory to be part of the fault data path. This change leads to much poorer performance than multi-GPU computing without oversubscription for three reasons. First, the host virtual memory system is slower in updating page tables and shooting down TLB entries. Second, the CPU-GPU interconnect may be slower than the GPU-GPU interconnect and takes longer to transfer the same amount of data. Third, eviction is on the critical path and blocks the migrations until VABlock space is available.

7.3 Effects of Advanced Interconnects on Demand Paging

The system must meet several conditions to fully benefit from advanced interconnect such as Nvidia's NVLink2 and AMD's Infinity Fabric. First, prefetching must be in place to transfer a large amount of data to saturate the bandwidth and amortize the cost. Second, the prefetched pages should effectively reduce page faults. Third, all interconnections in the fault data path must be equipped with advanced interconnects, because the weakest link limits the overall performance. These interconnections include those from the migration source to the destination and those from the eviction source to the destination.

Presently, UVM evicts pages to the host memory by default, no matter how many GPU devices are available on the system and whether the GPU-GPU interconnect is faster than the CPU-GPU interconnect. This default setting leverages the larger host memory capacity but may suffer from the low CPU-GPU bandwidth and high OS overhead. Adapting the eviction destinations to system configurations and workload characteristics may be an effective optimization.

7.4 Additional System Considerations

Our experimental environments have equal page sizes on the host and device sides. In an environment where host page size differs from device page size, the least common multiple will be used

in UVM. We do not evaluate such a system, but our methodology is not impacted by such changes and would still remain usable.

UVM currently applies to single-node communication. Several GPUs across a rack or cluster are managed by UVM locally, and each GPU in a node communicates with the UVM driver for unified memory. The next generation of NVLink is slated to introduce atomics across systems, implying that unified memory may span multiple nodes eventually. While this would certainly scale the reach of UVM, our analysis will still be relevant outside the context of any inter-node communication that would need to be added to UVM.

8 CONCLUSION

In this work, we perform fine-grain quantitative analyses of UVM demand paging and isolate the costs of tasks on the fault command and data paths among the host and GPU devices. By examining three settings of UVM page migrations with different source and destination devices and interconnects, we reveal synchronous and asynchronous tasks in UVM demand paging and discover key UVM performance factors and dominating costs. We find that the service component of the UVM driver is the dominating synchronous component, primarily fueled by page table operations such as unmapping source pages and remapping destination pages. We find that asynchronous components become a larger contributor in the presence of prefetching and oversubscription. We demonstrate how prefetching effectively reduces the cost of the dominating task, servicing, and how oversubscription changes the overlap between synchronous tasks and asynchronous tasks. In addition, we identify the benefits and constraints of inherent UVM features and hardware enhancements, such as NVLink allowing better overlap of synchronous and asynchronous tasks in the presence of prefetching.

ACKNOWLEDGEMENTS

The authors would like to thank the Advanced Computing Infrastructure team at Clemson for their support and the use of Clemson's Palmetto Cluster.

REFERENCES

[1] High Performance Geometric Multigrid. 2022. Retrieved July 13, 2021, from https://crd.lbl.gov/divisions/amcr/computer-science-amcr/par/research/hpgmg/

[2] Tyler Allen. 2022. *Holistic Performance Analysis and Optimization of Unified Virtual Memory*. Ph.D. Dissertation. Clemson University.

[3] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM cost with deep runtime and workload analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*. 141–150. https://doi.org/10.1109/IPDPS49936.2021.00023

[4] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*. Association for Computing Machinery, New York, NY, Article 64, 15 pages. https://doi.org/10.1145/3458817.3480855

[5] Nadav Amit. 2017. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC'17)*. USENIX Association, Santa Clara, CA, 27–39. https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[6] 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 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50'17)*. Association for Computing Machinery, New York, NY, 136–150. https://doi.org/10.1145/3123939.3123975

[7] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-software support for efficient page migration in multi-GPU systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 596–609. https://doi.org/10.1109/HPCA47549.2020.00055

[8] Natalie Beams, Ahmad Abdelfattah, Stan Tomov, Jack Dongarra, Tzanio Kolev, and Yohann Dudouit. 2020. High-order finite element method using standard and device-level batch GEMM on GPUs. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA'20)*. 53–60. https://doi.org/10.1109/ScalA51936.2020.00012

[9] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas R. W. Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC'19)*. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[10] Manuel Birke, Bobby Philip, Zhen Wang, and Mark Berrill. 2019. Block-relaxation Methods for 3D Constant-Coefficient Stencils on GPUs and Multicore CPUs. (2019). arXiv:cs.DC/1208.1975

[11] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos. *J. Parallel Distrib. Comput.* 74, 12 (Dec. 2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

[12] Chia-Hao Chang, Adithya Kumar, and Anand Sivasubramaniam. 2021. To move or not to move? Page migration for irregular applications in over-subscribed GPU memory systems with DynaMap. In *Proceedings of the 14th ACM International Conference on Systems and Storage*. 1–12.

[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *ArXiv* abs/1410.0759 (2014).

[14] Steven Chien, Ivy Peng, and Stefano Markidis. 2019. Performance evaluation of advanced features in CUDA unified memory. *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC'19)*. https://doi.org/10.1109/mchpc49590.2019.00014

[15] Linux Kernel Development Community. Heterogeneous Memory Management (HMM). 2023. Retrieved May 25, 2021, from https://www.kernel.org/doc/html/latest/mm/hmm.html

[16] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *High Performance Computing*, Michela Taufer, Bernd Mohr, and Julian M. Kunkel (Eds.). Springer International Publishing, Cham, 489–507.

[17] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *Int. J. High Perform. Comput. Appl.* 30, 1 (2016), 3–10. https://doi.org/10.1177/1094342015593158 arXiv:https://doi.org/10.1177/1094342015593158

[18] Debashis Ganguly, Rami Melhem, and Jun Yang. 2021. An adaptive framework for oversubscription management in CPU-GPU unified memory. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE'21)*. IEEE, 1212–1217.

[19] 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 46th International Symposium on Computer Architecture (ISCA'19)*. ACM, New York, NY, 224–235. https://doi.org/10.1145/3307650.3322224

[20] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2020. Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS'20)*. IEEE. https://doi.org/10.1109/ipdps47924.2020.00054

[21] Rahulkumar Gayatri, Kevin Gott, and Jack Deslippe. 2019. Comparing managed memory and ATS with and without prefetching on NVIDIA volta GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS'19)*. 41–46.

[22] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing large graphs on GPUs with unified memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1119–1133. https://doi.org/10.14778/3384345.3384358

[23] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. 2020. UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs. (2020). arXiv:2007.09822.

[24] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing memory in heterogeneous systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 637–650.

[25] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. 2005. An overview of the trilinos project. *ACM Trans. Math. Softw.* 31, 3 (Sept. 2005), 397–423. https://doi.org/10.1145/1089014.1089021

[26] John Hubbard and Jerome Glisee. 2017. GPUs: HMM: Heterogeneous Memory Management. https://www.redhat.com/files/summit/session-assets/2017/S104078-hubbard.pdf

[27] 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 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. https://doi.org/10.1145/3373376.3378529

[28] Marcin Knap and Paweł Czarnul. 2019. Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. *J. Supercomput.* 75 (Nov. 2019), 7625–7645. https://doi.org/10.1007/s11227-019-02966-8

[29] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. 2014. An investigation of Unified Memory Access performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC'14)*. 1–6.

[30] Michael Larabel. AMD Making Progress on HMM-based SVM Memory Manager for Open-Source Compute. 2021. https://www.phoronix.com/news/AMD-ROCm-HMM-SVM-Memory

[31] Bingyao Li, Jieming Yin, Anup Holey, Youtao Zhang, Jun Yang, and Xulong Tang. 2023. Trans-FW: Short circuiting page table walk in multi-GPU systems via remote forwarding. In *2023 IEEE International Symposium on High Performance Computer Architecture (HPCA'23)*.

[32] 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 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 49–63.

[33] Sheng Lin, Ning Liu, Mahdi Nazemi, Hongjia Li, Caiwen Ding, Yanzhi Wang, and Massoud Pedram. 2018. FFT-based deep learning deployment in embedded systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE'18)*. 1045–1050. https://doi.org/10.23919/DATE.2018.8342166

[34] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. Association for Computing Machinery, New York, NY, 1633–1649. https://doi.org/10.1145/3318464.3389705

[35] Karthik Vadambacheri Manian, A. A. Ammar, Amit Ruhela, Chinghsiang Chu, Hari Subramoni, and Dhabaleswar K. Panda. 2019. Characterizing CUDA unified memory (UM)-aware MPI designs on modern GPU architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs (GPGPU'19)*. ACM, New York, NY, 43–52. https://doi.org/10.1145/3300053.3319419

[36] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen mei Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. (2020). arXiv:2006.06890.

[37] Saiful A. Mojumder, Yifan Sun, Leila Delshadtehrani, Yenai Ma, Trinayan Baruah, José L. Abellán, John Kim, David Kaeli, and Ajay Joshi. 2020. MGPU-TSM: A Multi-GPU System with Truly Shared Memory. (2020). arxiv:2008.02300.

[38] Jose M. Nadal-Serrano and Marisa Lopez-Vallejo. 2016. A performance study of CUDA UVM versus manual optimizations in a real-world setup: Application to a Monte Carlo wave-particle event-based interaction model. *IEEE Trans. Parallel Distrib. Syst.* 27, 6 (2016), 1579–1588.

[39] NVIDIA. Open GPU Documentation. (????). Retrieved May 25, 2021, from https://nvidia.github.io/open-gpu-doc/

[40] Steve Plimpton. 1995. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.* 117, 1 (1995), 1–19. https://doi.org/10.1006/jcph.1995.1039 http://lammps.sandia.gov.

[41] Steve Plimpton. 2017. FFTs for (Mostly) Particle Codes within the DOE Exascale Computing Program. https://www.osti.gov/servlets/purl/1483229

[42] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I. Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated on-demand high-throughput storage access in the bam system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. Vol. 2, Association for Computing Machinery, 325–339.

[43] Nikolay Sakharnykh. 2016. High-performance Geometric Multi-Grid with GPU Acceleration. Retrieved May 25, 2021, from https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/

[44] Nikolay Sakharnykh. 2019. Memory Management on Modern GPU Architectures. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9727-memory-management-on-modern-gpu-architectures.pdf

[45] Chuanming Shao, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, and Minyi Guo. 2022. Oversubscribing GPU unified virtual memory: Implications and suggestions. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 67–75.

[46] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. 2010. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. Association for Computing Machinery, New York, NY, 253–262. https://doi.org/10.1145/1810085.1810120

[47] Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, et al. 2022. CARAT cake: Replacing paging via compiler/kernel cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 98–114.

[48] Stanimire Tomov, Azzam Haidar, Daniel Schultz, and Jack Dongarra. 2018. *Evaluation and Design of FFT for Distributed Accelerated Systems.* ECP WBS 2.3.3.09 Milestone Report FFT-ECP ST-MS-10-1216. Innovative Computing Laboratory, University of Tennessee. Revision 10-2018.

[49] Chao Yu, Yuebin Bai, and Rui Wang. 2021. Enabling large-reach TLBs for high-throughput processors by exploiting memory subregion contiguity. *CoRR* abs/2110.08613 (2021). https://arxiv.org/abs/2110.08613

[50] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, Hui Guo, and Zhiying Wang. 2020. Coordinated page prefetch and eviction for memory oversubscription management in GPUs. In *International Parallel and Distributed Processing Symposium (IPDPS'20).* IEEE, 472–482. https://doi.org/10.1109/IPDPS47924.2020.00056

[51] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. 2020. HPE: Hierarchical page eviction policy for unified memory in GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2461–2474.

[52] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. 2019. A quantitative evaluation of unified memory in GPUs. *J. Supercomput.* 76, 4 (Nov. 2019), 2958–2985. https://doi.org/10.1007/s11227-019-03079-y

[53] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16).* 345–357.

Received 31 March 2023; revised 14 August 2023; accepted 23 October 2023