Shared Virtual Memory: Its Design and Performance Implications for Diverse Applications

Bennett Cooper
bwc@clemson.edu
Clemson University
Clemson, South Carolina, USA

Thomas R. W. Scogland
scogland1@llnl.gov
Lawrence Livermore National Lab
Livermore, California, USA

Rong Ge
rge@clemson.edu
Clemson University
Clemson, South Carolina, USA

ABSTRACT

Discrete GPU accelerators, while providing massive computing power for supercomputers and data centers, have their separate memory domain. Explicit memory management across device and host domains in programming is tedious and error-prone. To improve programming portability and productivity, Unified Memory (UM) integrates GPU memory into the host virtual memory systems, and provides transparent data migration between them and GPU memory oversubscription. Nevertheless, current UM technologies cause significant performance loss for applications. With AMD GPUs increasingly being integrated into the world's leading supercomputers, it is necessary to understand their Shared Virtual Memory (SVM) and mitigate the performance impacts. In this work, we delve into the SVM design, examine its interactions with applications' data accesses at fine granularity, and quantitatively analyze its performance effects on various applications and identify the performance bottlenecks. Our research reveals that SVM employs an aggressive prefetching strategy for demand paging. This prefetching is efficient when GPU memory is not oversubscribed. However, in tandem with the eviction policy, it causes excessive thrashing and performance degradation for certain applications under oversubscription. We discuss SVM-aware algorithms and SVM design changes to mitigate the performance impacts. To the best of our knowledge, this work is the first in-depth and comprehensive study for SVM technologies.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; **Processors and memory architectures**; • **Software and its engineering → Memory management**; • **Hardware → Hardware accelerators**.

KEYWORDS

Unified Memory, Heterogeneous Memory Management, GPGPU

ACM Reference Format:
Bennett Cooper, Thomas R. W. Scogland, and Rong Ge. 2024. Shared Virtual Memory: Its Design and Performance Implications for Diverse Applications. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656608

1 INTRODUCTION

Discrete GPU accelerators, with their massive parallel processing capabilities and energy efficiency, are crucial for providing the computing power of today's HPC systems and data centers. They enable significant advancements in areas such as climate modeling [7], bio-informatics [38], and AI. As of today, 90% of top 10 and more than 35% of top 500 supercomputers are accelerated by discrete GPUs [41], providing about 50% of the performance share of top 500 supercomputers. They have also become the predominant hardware for deep learning and large language models (LLMs) training. For example, the BLOOM (176B parameters) is trained over 1 million GPU hours using BigScience infrastructure [29], and the GPT-3 (175B parameters) is estimated to be over several million GPU hours [8].

Discrete GPU accelerators have their own separate memory domains from the host memory domain. Programmers must invoke memory copy functions and ensure the data residing in the GPU doesn't exceed GPU memory capacity. Explicit memory management and data movement across domains are laborious and error-prone, especially for memory-demanding workloads where the memory footprint exceeds GPU memory. It is increasingly important to relieve programmers of such tasks and make GPU programming more productive and portable, as deep learning models and social networks are increasingly larger [8, 29, 35, 44], and scientific workloads are more data-intensive [7, 38].

Unified Memory (UM) integrates GPU memory into the host virtual memory systems and transparently migrates data between them. Additionally, UM supports *GPU memory oversubscription*, i.e., GPU kernels access more data than the GPU memory can hold, significantly enhancing programming portability and productivity for memory-demanding workloads. UM technologies have been adopted by HPC frameworks such as Raja [6], Kokkos [9], and Trilinos [16] for writing portable applications on today's and future's major HPC platforms, and by deep learning frameworks [12, 22, 34]. However, even with active research and improvement by vendors and research community [3, 18, 23, 42], current UM technologies cause significant, or even prohibitive, performance degradation [25, 26, 46].

To bridge the performance gap between explicit memory management and Unified Memory (UM), a deep understanding of UM's design and identification of performance bottlenecks are crucial. NVIDIA GPUs have been the primary choice for accelerators in supercomputers and data centers, sparking significant research interest in NVIDIA's Unified Virtual Memory (UVM). Researchers have examined UVM's design, its impact on application performance, and proposed optimization techniques [10, 13, 31]. In recent years, AMD GPUs have seen a notable growth in adoption, powering 7 of the 10 most energy-efficient supercomputers. Both the fastest supercomputer, Frontier, and the upcoming leader, El Capitan [28], utilize AMD GPUs for acceleration. However, AMD's UM technology, Shared Virtual Memory (SVM), hasn't received much research attention.

SVM has a distinct design from UVM, and the insights derived for UVM may not be directly applicable to SVM. While both are implemented as software drivers mirroring page tables on both the host and the device, UVM is a technology developed for NVIDIA's specific hardware and drivers for performance, optimization, and efficiency. In contrast, SVM interfaces with the Linux kernel's Heterogeneous Memory Management (HMM) [19, 32], which is designed for broader hardware compatibility and integration. While HMM is still in development, its unified framework is poised to greatly simplify driver development and enhance application portability.

In this work, we delve into the design of the shared Virtual Memory (SVM) and examine its impact on the performance across a variety of applications. We investigate its UM management strategy, architecture and components, page fault handling and data migration/eviction in demand paging. We further quantify its overhead and the overall cost, and identify its performance bottlenecks and its variations with applications' data access. Using the fine-grain fault and migration profiles, we classify the applications and their access patterns, and reason the root causes of performance bottlenecks.

We reveal that SVM manages the unified memory by ranges, i.e., a range is a number (typically large) of contiguous pages. With demand paging, a single fault can trigger an entire range migration, which in turn requires a range eviction if GPU memory is oversubscribed. *This management strategy is equivalent to the most aggressive prefetching.* We find that the current SVM design is beneficial if the GPU is not oversubscribed, but otherwise causes excessive thrashing and performance degradation for applications with certain temporal and spatial access patterns. Due to limited information available, the eviction policy may evict the most intensely reused data, further exacerbating thrashing. Our quantitative analyses uncover that severe thrashing not only increases the eviction-to-migration ratio, but more seriously increases the number of migrations by orders of magnitude for certain applications. We establish that SVM-aware algorithm designs can significantly improve performance, and discuss possible augmentations to SVM design that could benefit broader applications.

We make the following main contributions:

- We reveal the SVM design and range migration/eviction in demand paging, and quantitatively analyze the UM management overhead and the overall costs at fine granularity. We identify the performance bottlenecks, their significant increases under oversubscription, and their variations across a variety of applications.
- We unveil the migration and eviction profiles and fault behaviors of diverse applications resulting from the interaction between their memory access and SVM. These profiles are indicative of performance and expose premature evictions and severe thrashing in applications with intensive data reuse or distributed data accesses.

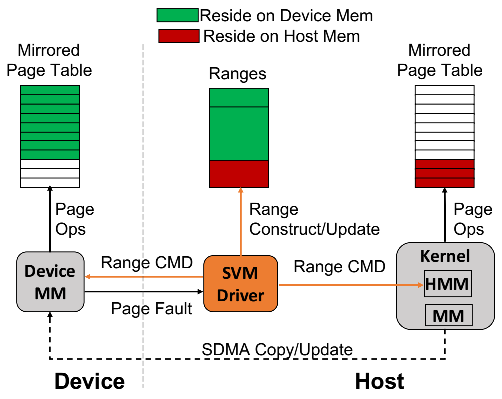


Figure 1: SVM manages the UM space by ranges, rather than pages in host and device memory domains

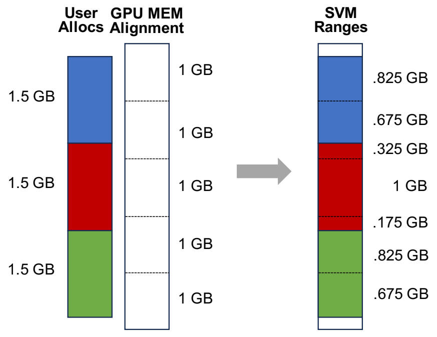


Figure 2: Example range creation for three 1.5 GB allocations

- We investigate potential benefits from SVM-aware application algorithm design using case studies. We show that SVM-awareness empowers us to mitigate performance bottlenecks and improve performance by up to orders of magnitude. We further discuss potential augmentations to SVM designs to benefit a broader range of applications.
- To the best of our knowledge, this work is the first in-depth and comprehensive study of SVM technology. As various accelerators and devices are expected to interface with Linux HMM, this work may shed light on optimal driver-specific designs and library implementations for target applications.

2 SVM DESIGN AND ARCHITECTURE

Unified Memory (UM) integrates the device memory domain into the host's virtual memory system, providing a shared address space for host processes and GPU kernels. UM transparently migrates data between the two memory domains, and keeps track of the physical memory locations on page tables, eliminating the need for programmers to copy explicitly. UM supports oversubscription by evicting old pages from GPU memory before migrating new ones.

Shared Virtual Memory (SVM) driver interfaces with Heterogeneous Memory Management (HMM) to interact with host page tables. HMM is expected to be the standard Linux interface for various driver modules. Unlike the one-way communication from the older drivers to the kernel, HMM creates a truly unified memory by preventing the kernel from moving pages without alerting the driver and causing many edge cases of failures.

In this section, we detail SVM's design and architecture. Our experimental platform is one node from the LLNL Tioga supercomputer [27]; the node architecture matches those of the Frontier supercomputer [33, 37, 40]. A node consists of a 64-core AMD 7A53 EPYC CPU, 512 GB DDR4 host memory, and four AMD Instinct MI250X discrete GPUs connected to the host by 36GB/s bidirectional Infinity Fabric. Each MI250X has two GPU compute dies connected by 200GB/s bidirectional Infinity Fabric, each with 64 GB HBM2E memory. Tioga uses the Tri-Lab Operating System Stack [30] version 4 & amdgpu version 6.3.6, with 1 GB GPU memory alignment in SVM. We use ROCM version 5.4.0. The experimental results presented in this work only use one GPU compute die.

2.1 Ranges as SVM Management Units

Even though both host and device memory domains manage their own address space in pages, SVM manages the unified memory in **ranges**, as shown in Figure 1. Each SVM range is defined by a start address and an end address and may comprise a substantial number of contiguous virtual pages. The management operations include allocation and deallocation, migration, and eviction.

Upon the receipt of a managed memory allocation from the runtime, SVM constructs the ranges based on GPU memory alignment and the allocation's size. GPU memory alignment is determined by its capacity, i.e., $\lfloor \frac{\text{capacity}}{32} \rfloor$ rounded down to the nearest power of two, and should be minimally 2 MB. For example, if a GPU has 48 GB available for SVM managed memory, then the alignment is 1 GB. In addition, the ranges must be aligned to allocation boundaries. With this range construction, an allocation should comprise multiple ranges if it is large or across alignments.

Figure 2 depicts the range construction for an application with three 1.5 GB allocations on a GPU aligned by 1 GB. The application mimics matrix multiply. SVM constructs 7 ranges of varying sizes for this application, with the smallest range at 175 MB and the largest at 1 GB.

SVM receives faults at the page level from the device but services with data migration at the range level. While a range may consist of up to 256K pages, it only requires the servicing of a single page fault to trigger the migration of the entire range, with the remaining faults being dismissible. Thus, a received fault undergoes an initial examination to determine if it is serviceable.

A fault is considered serviceable if it is recent and not duplicate. Recent faults are those with timestamps falling within the specified timeout period. Old unsatisfied faults would be replayed by the GPU to generate recent faults. A recent fault is considered a duplicate if it originates from the same page or range as a recent range migration. Duplicate faults typically dominate, representing 97-99% of the total faults generated by a kernel. They can arise from various sources, including the same thread block processing data with spatial locality, different thread blocks processing the same data, and the same thread block processing data with temporal locality. In practice, applications with a high degree of duplication of faults can perform efficiently by consolidating them into a single range transfer.

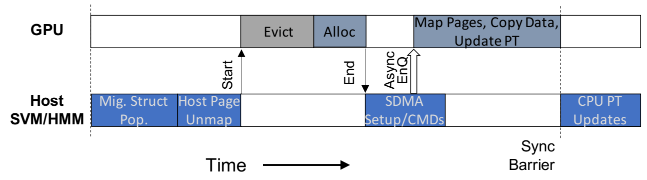


Figure 3: Timeline of range migration for a serviceable fault. "Evict" only occurs if there is insufficient space for "Alloc".

2.2 Page-Level Fault and Range-Level Migration

Each serviceable fault triggers SVM to migrate the range in which the faulting page is. SVM keeps track of the residency of each range and uses it to determine the migration direction (e.g., host-to-device, device-to-host, or device-to-device). Due to page limits, we focus on the host-to-device migration as it is the most important in GPU computing.

To migrate a range from host to device, the SVM driver provides HMM the start and end addresses of the range and obtains a list of source physical frame numbers (PFNs) necessary for data migration. HMM further leverages the built-in memory management in the Linux kernel for host memory management and page table operations.

Figure 3 shows the timeline of the host-to-device migration visible in the SVM driver. Essentially, the SVM driver sends commands to the host and the device and synchronizes their operations. The commands to the host/HMM include obtaining source PFNs, and performing page unmapping and page table updates, and commands to the device include allocating ranges and pages, initiating direct memory copy, and performing page mapping and page table updates. Note that the System Direct Memory Access (SDMA) copy is asynchronous and used for page content copy, GPU page mapping and unmapping, and page table updates. The associated cost partly overlaps with SDMA setup and command issuing on the SVM driver.

During the GPU memory allocation, if there isn't enough available space for the range to be migrated, SVM must first evict one or more residing ranges, as shown in Figure 3. SVM employs the least recently faulted (LRF) policy to determine the next victim range and continues to evict ranges until the available space becomes sufficient. It's important to note that eviction is costly, involving various operations such as page mapping, unmapping, and content copying as in migration, albeit in the opposite direction. We show the quantitative costs in detail in section 2.4.

2.3 Overall SVM Architecture

Figure 4 presents the SVM architecture, and how its modules interact to service a serviceable page fault originating from a compute unit (CU). A page fault occurs if address translation using TLBs and page tables fails. Upon the fault, the L2 TLB sends a "translation negative acknowledgment" (XNACK) back to the CU and writes an interrupt cookie to an on-device buffer Content-Addressable-Memory (CAM) ①, in which faults on the same pages can be filtered. The interrupt controller reads from the buffer ② and passes these cookies along to the SVM driver for servicing ③. Meanwhile, the CU retries the faulted access until translation succeeds.

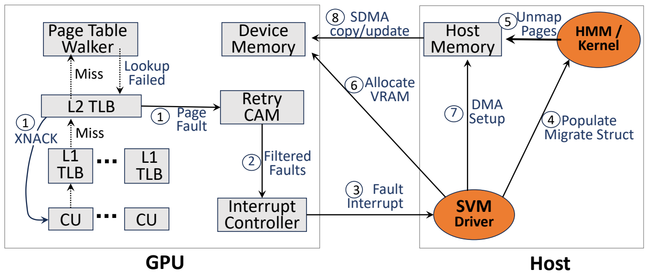


Figure 4: SVM Architecture. The steps are components' interactions in response to a page fault originating from a compute unit (CUs).

The SVM driver running on the host decodes the interrupt cookie. This cookie contains the faulting page address, the timestamp, and the access type. The driver determines if the fault is serviceable, i.e., neither timed out nor a duplicate range.

For a serviceable fault, SVM creates a `migrate_vma` structure and passes it to HMM along with the corresponding range in order to get the host source page PFNs populated ④. HMM performs a page table walk over the range and unmaps all pages on the host ⑤. The SVM driver then allocates memory on the GPU device as needed for the migration ⑥. Next, the driver sets up the System Direct Memory Access (SDMA) mapping and issues commands ⑦ to asynchronously copy data, perform paging, and update GPU page tables ⑧. Migration concludes at the synchronization point.

SVM architecture shows distinct strategies for fault handling and unified memory management compared to NVIDIA's Unified Virtual Memory (UVM), which has garnered relatively more research attention [1, 2, 24]. Table 1 presents the key differences. Unlike UVM which batches page faults in a buffer and handles the batch when the buffer is full, SVM receives a single fault each time and handles it immediately. Such a strategy has two main advantages. First, serviceable faults are serviced immediately to reduce the turnaround time for individual accesses. Second, duplicate page faults are quickly identified and dismissed from servicing. The main disadvantage is that the SVM driver is heavily loaded with fault interrupts, even after some faults are filtered in the CAM buffer on the GPU side.

SVM manages the UM at the range granularity for allocation, deallocation, migration, and eviction. While being varying sizes, a range is typically orders of magnitude larger than a VABlock (2 MB) used in UVM. If all migrated data are to be used, migrating an entire range effectively amortizes data access latency and fully utilizes the host-device interconnect bandwidth. The range granularity is beneficial for scenarios when GPU memory is not oversubscribed or for applications whose data are not evicted before use. Otherwise, it causes significant performance issues for two main reasons. First, GPU memory gets exhausted faster with data not immediately needed, requiring evictions to make space for subsequent migrations. Second, it causes severe thrashing, i.e., data migrated but evicted before use must be migrated again. Thrashing degrades performance in two ways: wasting time in migrating and then evicting unused data, and more importantly increasing the migration frequency. Case studies of severe thrashing are presented in Section 3.4.

UM Feature	SVM	UVM
Fault batching	No	Yes
Fault handling	Single fault	Fault batch[a]
UM (De)alloc	Range (∈[4KB, 1GB])	VABlock (2MB)
Migration unit	Range	Page[b]
Eviction unit	Range	VABlock
Eviction Policy	Least Recently Faulted	

[a] A batch consists up to a system-configurable 256 faults.
[b] 64 KB without prefetching, and up to a VABlock with prefetching.

Table 1: SVM vs. UVM.

2.4 SVM UM Management Costs

We quantitatively analyze the cost for SVM UM management. We use Systemtap [39] to dynamically instrument and trace the SVM driver functions and events. We run each instance twice for data collection: first to measure the timing of SVM driver functions, and then to capture events such as faults, migrations, and evictions.

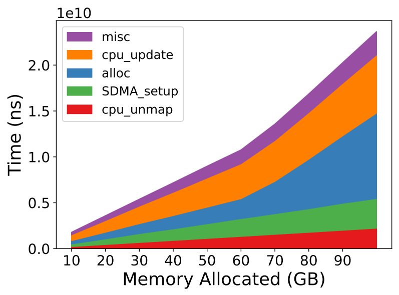
Here we focus on the major cost items during fault servicing and corresponding migration, based on Figure 3, and ignore others including fault receiving, preprocessing, and filtering as their costs are relatively small and negligible.

- **cpu_unmap**: collect and unmap host pages.
- **SDMA_setup**: create SDMA mappings, and issue SDMA commands to perform copy, mapping, and page updates.
- **alloc**: allocate physical VRAM on the device. Note that this item includes the cost of eviction if there is insufficient space for allocation.
- **cpu_update**: update CPU page table with new mappings if migration succeeded or restore old mappings if failed.
- **misc**: migrate page meta-data, non-overlapped SDMA copy, and free copy mappings.

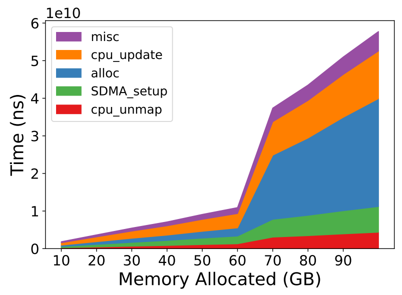
cpu_unmap and **cpu_update** manage pages and page tables on the host side, and the actual data movement is encapsulated in **SDMA_setup** and **misc**. All these costs are visible on the host and are reflected in the application's execution time.

Figure 5 shows the cost items for three representative applications over various problem sizes ranging from small to large enough to oversubscribe GPU memory by 56%. Note that each cost item is accumulated over all the migrations in the kernel. From these figures, we make some key observations.

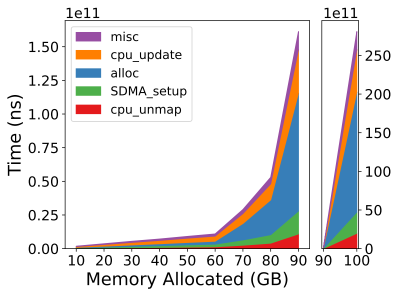
- The total cost increases with problem size as expected because the number of migrations increases. However, the applications exhibit distinct growth trends. STREAM displays two linear segments separated by oversubscription, with the slope of the second segment being slightly larger. Both Jacobi2D and SGEMM present three or more segments. For Jacobi2D, the second segment's slope is the largest where GPU memory is oversubscribed by less than 10%. For SGEMM, the last segment's slope is significantly larger, surpassing the others by orders of magnitude.
- For small problem sizes without oversubscribing GPU memory, **cpu_update** is the largest individual component, followed by **SDMA_setup** and **alloc**. These three account for roughly 76% of the overall cost for all three applications.
- While all cost items increase under oversubscription, **alloc** increases the most and becomes dominant across the applications. This increase is due to the evictions for freeing GPU



(a) STREAM



(b) Jacobi2d



(c) SGEMM

Figure 5: The cost of SVM UM management and range migration. SGEMM is shown in two windows as the magnitude of the second visually erases the first.

Benchmarks	Description	Domain	Source
STREAM	Triad-only. Scaled dot product of two vectors.	Synthetic	RAJAPerf [17]
Conv2d	Full convolution in a 2D space with varying weights.	Machine Learning	RAJAPerf [17]
Jacobi2d	Forward then backwards adjacent convolution with equal weights.	Machine Learning	RAJAPerf [17]
BFS	Breadth First Search graph traversal from randomly selected node.	Graph Traversal	EMOGI [35]
SYR2K	Symmetric rank-2k update from ROCBLAS	Linear Algebra	rocBLAS [4]
SGEMM	General matrix-matrix product from ROCBLAS	Linear Algebra	rocBLAS [4]
MVT	Matrix-vector product followed by matrix-transpose-vector product.	Linear Algebra	RAJAPerf [17]
GESUMMV	Sum of two scaled matrix-vector products.	Linear Algebra	RAJAPerf [17]

Table 2: Diverse benchmarks from multiple domains.

memory. Eviction comprises all other items in the opposite direction, thus is costly. The slightly larger slope of the second segment in STREAM suggests only a small number of evictions, while the drastically larger slopes in Jacobi2D and Sgemm suggest higher numbers of evictions.

These results indicate significant overhead for UM and demand paging. When GPU memory is not oversubscribed, the actual data movement across the host and device memory domains only accounts for less than half of the overall cost, and is smaller than the sum of UM management items including host and device mapping and unmapping and page table updates. More seriously, the UM management overhead increases substantially once GPU memory is oversubscribed, and becomes extremely high for applications such as SGEMM. We examine how the overhead impacts the performance of various applications in the following section.

3 WORKLOAD PERFORMANCE AND PROFILES

We examine diverse GPU workloads and study how their performances vary with GPU memory oversubscription. We further inspect their migration and eviction profiles, and use them to explain the performance change. The applications represent multiple domains as listed in Table 2. Some are directly from AMD ROCm implementations (e.g., rocBLAS SYR2K and SGEMM), and others are ported from RAJAPerf [17] implementations using Heterogeneous-computing Interface for Portability (HIP) APIs. We modify responding allocations to utilize managed memory. We have examined over a dozen applications but have only included those with complete data across different problem sizes.

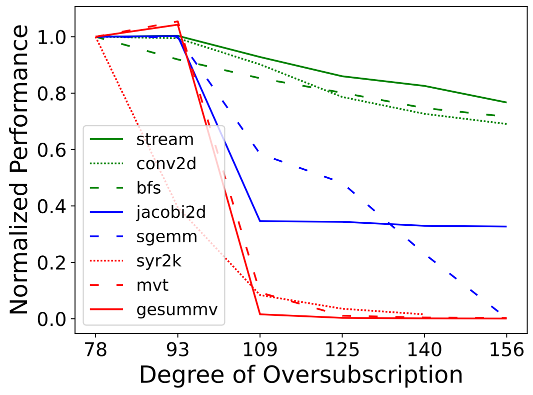


Figure 6: Performance decreases with the degree of oversubscription for various applications under SVM.

3.1 Performance Impacts of Oversubscription

We use the term Degree of Oversubscription (DOS) to quantify how much memory is used beyond the GPU's available capacity for unified memory (UM). DOS is defined as $used\_size/available\_size \times 100$. Based on this definition, a DOS value exceeding 100 indicates GPU memory oversubscription.

Figure 6 shows how performance varies with DOS with demand migration across the applications. Application performance is measured using throughput, which can be compute rate (FLOPs per Second) or memory throughput (GBs per second), and is normalized to that at DOS = 78. We use normalization here to emphasize the change in an application's performance relative to the problem size. While all applications' performances decrease monotonically as DOS increases, they exhibit different patterns. We group the patterns into three categories.

Category I: Performance declines moderately as DOS increases. STREAM and Conv2d belong to this category, though their rates of decrease are different. When oversubscribing GPU memory, applications in this category experience the least impact with demand migration.

BFS appears to be in category I, warranting explanation. BFS's execution depends on the input graph and the start node. Our case study uses a randomly generated graph with 10% of possible edges and a randomly selected start node. While accesses to nodes and edges are expected to be random, this randomness is confined within ranges, and the accesses across the ranges still follow a linear pattern.

Category II: Performance declines significantly once DOS surpasses 100, and then minimally changes thereafter. Jacobi2D belongs to this category and its performance decreases to about 40% at $DOS = 109$.

Category III: Performance drops close to zero when DOS surpasses 100 or more. The decline can be abrupt as seen in GESUMMV and MVT, or gradual as in SGEMM. When oversubscribing, applications in this category experience the most impact using the SVM on-demand migration.

What are the factors responsible for the varying performance differences among the applications as DOS changes? To answer this question, we inspect the migration and eviction profiles resulting from the applications' interaction with SVM. Limited by the long time needed to gather experimental results, Figure 6 only displays performance data for DOS values up to 156. Other questions that naturally arise are: What are the performance of applications like STREAM and Jacobi2D as DOS continues to increase? We establish the answers using insights from fine-grain application profiles presented next.

3.2 Migration and Eviction Profiles

The performance impact is determined by the complex interplay between the application's memory request and SVM UM management. As noted before, performance degradation under oversubscription is primarily attributed to eviction and, even more significantly, to thrashing. Eviction is on the critical path, meaning eviction is only initiated by the migration request in the opposite direction, which is blocked until eviction is completed. Eviction doubles the cost of migration and delays the migration and computation. In general, the more an application evicts, the larger the performance loss it suffers.

In an on-demand migration memory model, eviction is inevitable for problem sizes that exceed the GPU's physical memory. While eviction directly results in performance loss, some evictions are more costly. Eviction has two types: permanent eviction, which displaces data no longer needed, and premature eviction, which displaces data required for current or future computation. Permanent evictions simply increase migration costs, while premature evictions further lead to thrashing, which has a compounding effect by increasing the eviction-to-migration ratio and the number of migrations.

Premature evictions occur in applications with certain temporal and spatial access patterns. The temporal pattern involves data reuse, and particularly the repeated traversal of one or more memory allocations. Such a pattern is commonly found in algorithms with nested loops, such as BLAS-2 and BLAS-3 algorithms. Any eviction of the repeatedly traversed allocations is premature, necessitating the subsequent migration. The spatial pattern involves successive accesses of a small amount of data that is distributed across the ranges of the same allocations. Applications displaying such patterns rapidly fill the GPU's memory, leading to frequent evictions, of which a significant portion is premature.

Figure 7 shows migration and eviction profiles at $DOS = 109$ across the applications. The data are collected using Systemtap as described in Subsection 2.4. It is worth noting that the profiles only reveal partial information, i.e., transfers of ranges across host-device interconnect that involve the SVM driver. They are unable to show access to data within the ranges or data re(use) on the GPU device. Missing information such as the number of faults a migration satisfies is crucial for identifying performance bottlenecks and opportunities for optimization. We discuss it in detail in Subsection 3.3.

Applications in Category I such as STREAM and Conv2d involve only permanent evictions. The ranges of each allocation are migrated in a linear streaming fashion, and all allocations are concurrently accessed. Once GPU memory is oversubscribed, ranges migrated the earliest are evicted successively. These applications don't have data reuse.

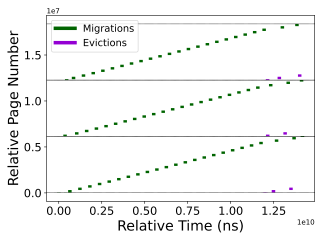
BFS iterates over multiple GPU kernels using the same data and thus incurs premature evictions. Determined by the linear traversal of the edge list's ranges and the minimal computation, BFS's degradation complies more with category I.

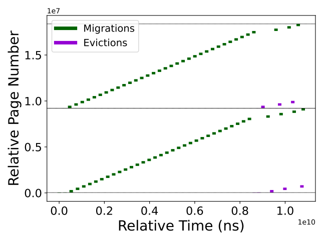
Applications in Category II such as Jacobi2D also exhibits linearly progressed range migrations over all of its allocations. However, Jacobi2D experiences premature evictions: ranges are evicted and then re-migrated a short time later. Jacobi2d iterates the same data accesses and computations, and Figure 7d illustrates two iterations. Execution with a larger problem size should have the same migration and eviction profiles but with an earlier onset of eviction in the initial iteration.

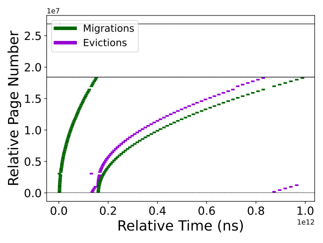
There are two subtypes of applications in Category III. One type includes applications SGEMM and SYR2K that exhibit linearly progressed range migrations and premature evictions for allocations, similar to Jacobi2d. A key feature is that their prematurely evicted data are intensively reused for computation at present and in the future, manifested by immediately re-migrating the same ranges after evicting them.

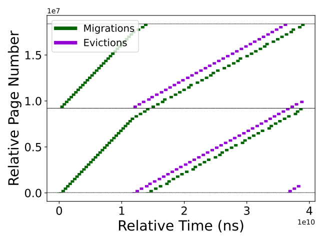
The other subtype in Category III such as MVT and GESUMMV display spatial patterns where successive data accesses are dispersed across the allocations as in matrix transpose. A large number of unique ranges are migrated and evicted simultaneously. GPU memory is quickly filled and evictions occur very early in the application's execution. Although MVT and GESUMMV do not reuse data as intensively as Sgemm and SYR2K , they experience similar thrashing. GESUMMV suffers more thrashing than MVT with two large allocations instead of one.

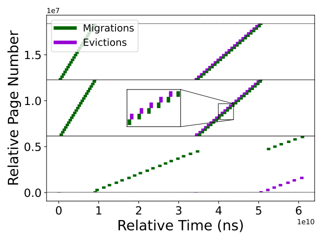
What is the performance of applications like STREAM and Jacobi2D as DOS continues to increase? We derive that STREAM's

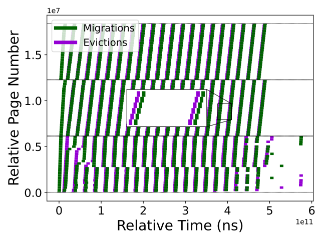

(a) STREAM

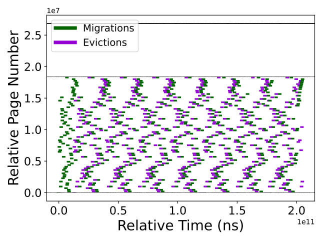

(b) Conv2d


(c) BFS


(d) Jacobi2d


(e) SGEMM


(f) SYR2K


(g) MVT

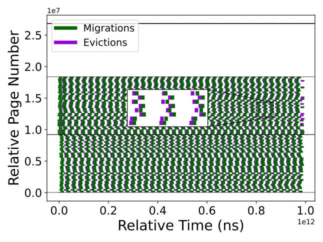

(h) GESUMMV

Figure 7: Migrations and evictions over execution time at $DOS = 109$. The y-axis is subdivided by allocation boundaries. White spaces are respective to smaller allocations. Though invisible, they also experience migration and evictions. Their invisibility is a result of the presentation: enlarging these data points causes the currently visible ones to occupy a solid-filled space.

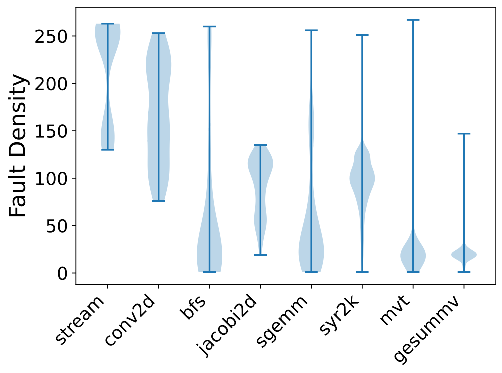


Figure 8: Overall fault densities for application executions with problem sizes at $DOS = 109$.

performance asymptotically approaches half of the highest performance when $DOS < 100$. As DOS increases, every migration, except the ones before the GPU memory is fully occupied, requires an eviction, which involves the same operations in the opposite direction. When the eviction-to-migration ratio approaches 1, performance halves. Using the same analysis, we derive that Jacobi2D's performance approaches 0.36.

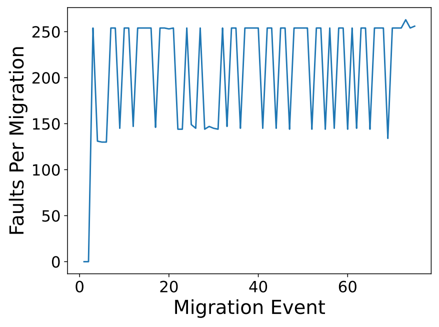
3.3 Fine-Grain Fault Behaviors

Migration and eviction profiles offer incomplete information and cannot entirely quantify the performance differences among the applications. Here, we delve into the detailed fault behaviors and assess how effectively they are handled by migrations.

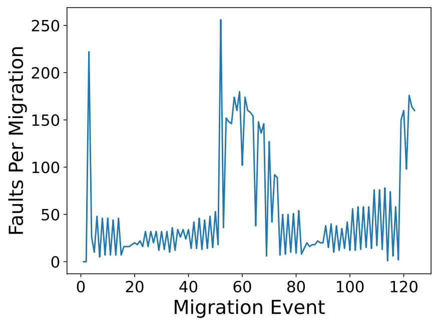
We use ***fault density*** to refer to the number of faults that are satisfied by a given migration. Here faults include both serviceable and dismissed to reflect the data request of applications. The higher the fault density is, the more effective a migration is. There are two conditions for an application to obtain high fault density. First, the application must consecutively request a large amount of data in the same range. Second, these accesses must occur simultaneously or in a small enough time frame, e.g., the time taken to service a fault. Applications with linearly progressed access inherently meet the first condition, but other access patterns may also meet it. The second condition is predominately driven by the arithmetic intensity, typically measured by the compute per data access [43], of an application. Greater arithmetic intensity results in lower fault density by enlarging the time window between accesses.

Figure 8 presents the fault densities across the applications. Overall, applications in Category I such as STREAM and Conv2d have the highest fault densities. They both have linearly progressed accesses. Between them, Conv2D has a somewhat lower fault density with its higher arithmetic intensity. Next comes Jacobi2d in Category II. While Jacobi2d also has linearly progressed accesses, it involves evictions which enlarges the time frame for the same number of faults. Applications in Category III such as MVT and Gesummv have the lowest fault densities for their successive accesses are distributed over the ranges. BFS is an exception, with linearly progressed accesses and low arithmetic intensity but a very low average fault density as those in Category III. This is explained by its random and sparse accesses of the edges and nodes within the ranges.

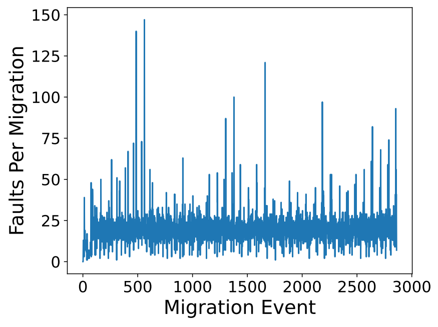
Each application's fault density has a certain range and distribution. Figure 9 shows how the fault density varies over time (Figure 9a-c) and over allocations using three applications (Figure 9d-f). STREAM's fault density largely falls in [150, 250] over time. SGEMM has a lower average fault density below 50 for it is computationally intensive. The spikes correspond to periods during which data migration occurs without concurrent computation.



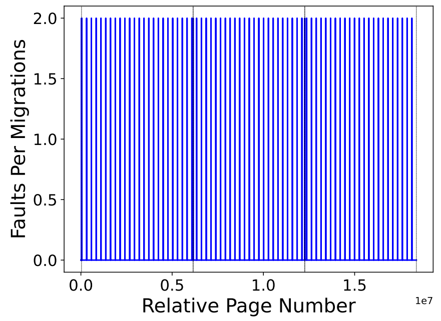
(a) STREAM



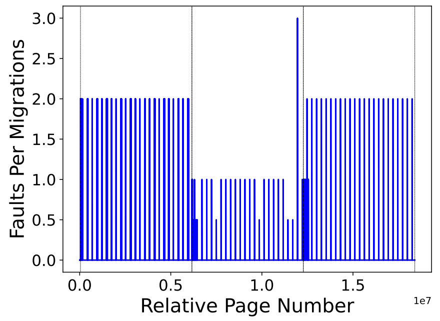
(b) SGEMM



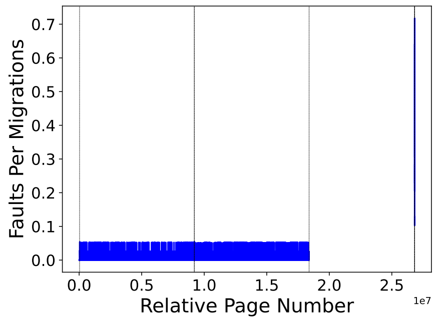
(c) GESUMMV



(d) STREAM



(e) SGEMM



(f) GESUMMV

Figure 9: Variation of fault density over time (a-c) and over allocations (d-f) at $DOS = 109$.

GESUMMV's fault density varies over time and fluctuates around 20 because it successively accesses to data distributed over ranges.

As shown in Figure 9d-f, only specific page numbers have a non-zero fault density and trigger migrations, while others encounter zero faults because their requests are already satisfied by the range migrations. Migration-triggering pages in STREAM and SGEMM are uniformly distributed across their allocations, complying with their linearly progressed data accesses. These pages correspond to those located at the start of the ranges. Note that each bar encapsulates a large number of such pages. The average faults per migration is 2, indicating duplicate faults. Migration-triggering pages in GESUMMV are not only more densely distributed but also have significantly lower faults per migration, approximately 0.05, or 20 migrations vs a single fault due to severe thrashes. This is explained by scarce and distributed successive data requests in GESUMMV.

3.4 Thrashing and Escalated Costs

Applications in Category III encounter thrashes, and their frequency increase with the degree of oversubscription. Here we study impacts of thrashing and associated escalated costs.

Figure 10 shows the increases of the eviction-to-migration ratio and migration counts with DOS. The eviction-to-migration ratio is 0 at $DOS < 100$ across the applications except BFS which algorithmically transfers data from the device to the host. The ratio quickly increases to 1 for applications in Category III, but only gradually increases for other applications, especially for application in Category I. A higher ratio corresponds to a higher cost per migration, and a ratio of 1 indicates each migration involves an eviction and thus doubled cost per migration.

The most severe performance degradation results from the order of magnitude increase in the migration counts. As DOS increases, the counts for applications in Category III increase drastically by an order of magnitude or more, and increase exponentially for SGEMM and SYR2K once DOS reaches 140. In contrast, the counts for STREAM and Conv2d only increase linearly, doubling when DOS doubles. The count for Jacobi2d initially exhibits a jump and then proceeds to increase linearly.

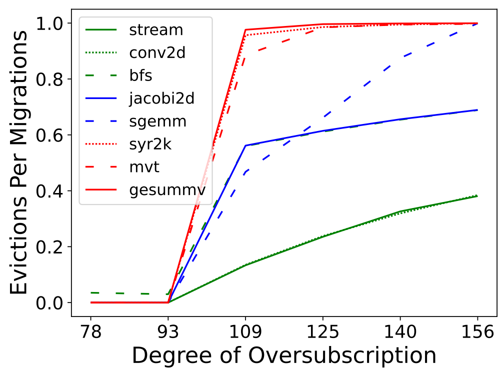
4 CONSIDERATIONS AND DISCUSSIONS

We discuss how application performance can be improved with SVM-aware algorithms and potential changes in SVM design.

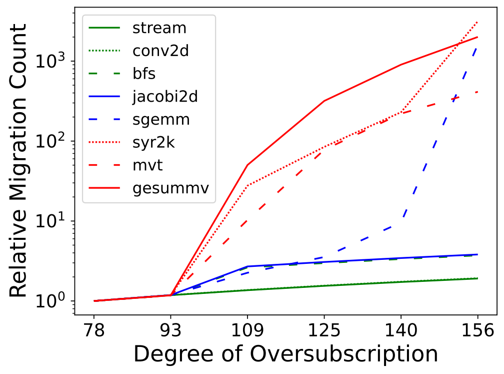
4.1 SVM-Aware Algorithm Design

Understanding the SVM design empowers us to identify the performance bottlenecks in existing algorithms and redesign them to optimize their interactions with SVM. In this section, we use SGEMM and Jacobi2d as case studies to demonstrate SVM-aware algorithm design. These case studies do not require changes to the current SVM design and parameters.

Jacobi2d's performance suffers in oversubscribed problem sizes due to an excessive amount of thrashing that is more than necessary. Jacobi2d iterates over two consecutive GPU kernels involving two matrices, each performing a partial convolution over one matrix to update the other, shown in Algorithm 1. Both kernels traverse matrix data elements in the same manner, i.e., from the first to the



(a) Eviction-to-Migration Ratio



(b) Escalated Migrations

Figure 10: Performance impacts of thrashing. The number of migrations are normalized to those at $DOS = 78$

Algorithm 1 Original Jacobi2d GPU Implementation

```
GPU_Kernel1:
    $B[i, j] \leftarrow 0.2 \times (A[i, j] + A[i-1, j] + A[i+1, j] + A[i, j-1] + A[i, j+1])$
GPU_Kernel2:
    $A[i, j] \leftarrow 0.2 \times (B[i, j] + B[i-1, j] + B[i+1, j] + B[i, j-1] + B[i, j+1])$
```

Algorithm 2 SVM-Aware Jacobi2d GPU Implementation

```
GPU_Kernel1:
    $B[i, j] \leftarrow 0.2 \times (A[i, j] + A[i-1, j] + A[i+1, j] + A[i, j-1] + A[i, j+1])$
GPU_Kernel2:
    $A[N-i, M-j] \leftarrow 0.2 \times (B[N-i, M-j] + B[N-i-1, M-j] + B[N-i+1, j] + B[N-i, M-j-1] + B[N-i, M-j+1])$
```

last row and from left to right within each row. Under oversubscription, the first kernel evicts the first rows needed at the beginning of the second kernel execution, while the second kernel then progressively evicts the later rows needed soon by itself. Consequently, each range undergoes premature eviction and thrashing.

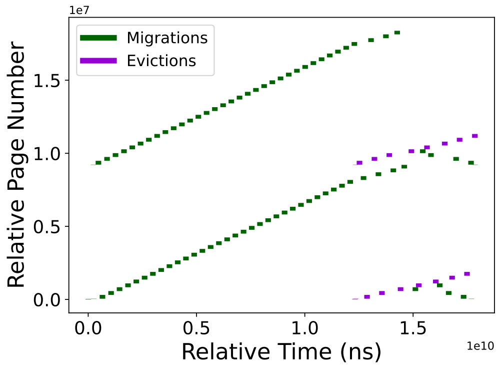


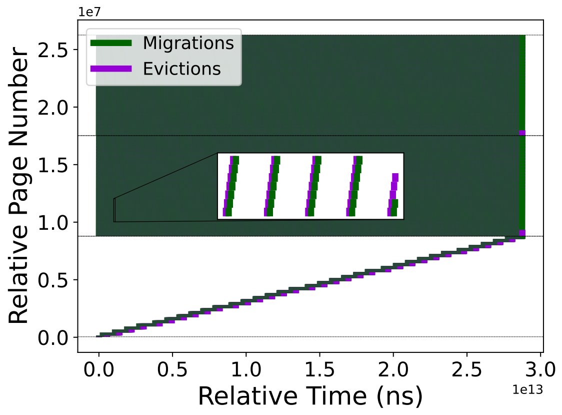
Figure 11: Timeline of migrations and evictions of the SVM-aware Jacobi2d implementation for small oversubscription. Compared to Figure 7d, thrashing is reduced significantly.

In the SVM-aware algorithm, we adjust the traversal order of the second kernel, i.e., from the last to the first row, and from right to left within each row, as in Algorithm 2. Such an adjustment allows the next kernel to fully reuse data residing on the GPU memory in the same iteration and across the consecutive iterations. Figure 11 shows the migration and eviction timeline for the SVM-aware Jacobi2d implementation. Compared to the timeline of the original implementation, shown in Figure 7d, we see significantly less evictions needed for the same computation.

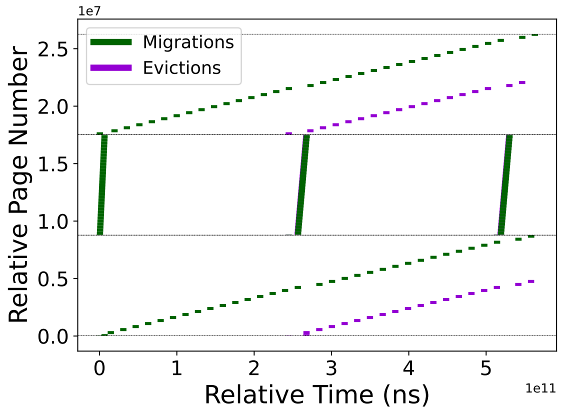
SGEMM's performance drops to 0% of relative performance. While the source code is unavailable to the public, its migration and eviction profiles in Figure 7e indicate that SGEMM first simultaneously migrates the entire allocation of each factor matrix, and then computes the product matrix row by row. As the number of computed rows of the product matrix is large enough to fill the GPU memory, new rows to be computed cause the eviction of factor matrix elements currently needed in computation. At this point, computation halts to re-migrate the newly evicted factor elements, which causes the eviction of the remaining factor elements. This chain of thrashing over factor matrix elements continues until the computed product rows become the least recently faulted and are evicted to make space for the new product rows. We speculate that SGEMM computes the total sum for a single product element at once by using an entire row of one factor to multiple an entire column of the other.

SGEMM is not scalable to support large problem sizes. For large problem sizes that cannot simultaneously fit both factor matrices in the GPU memory, the ranges of both factors are in a constant state of thrashing, depicted in Figure 12a.

We design a naive but SVM-aware GPU SGEMM implementation *SGEMM-svm-aware*, solely for the purpose of demonstrating the benefits of SVM-awareness instead of ultimate performance optimization. SGEMM-svm-aware migrates the entire column factor matrix to the GPU, and assigns a GPU thread block with a factor sub-matrix and the corresponding product sub-matrix to compute partial sums of the product elements. The total sums are carried out across the thread blocks. Computation only needs a chuck of rows for the row factor and the product at a time, and progresses over the rows. SGEMM-svm-aware significantly reduces the amount of thrashing, as shown in Figure 12b. Specifically, only one factor



(a) SGEMM



(b) SGEMM-svm-aware

Figure 12: Migration and eviction profiles of SGEMM and SGEMM-svm-aware at *DOS* = 156.

matrix (middle allocation) experiences thrashing twice, and the others encounter permanent evictions only.

Figure 13 presents the overall performance improvement using the SVM-aware algorithms. These algorithms show advantages over the counterparts in two aspects: preventing sudden drops in performance and elevating the lower performance limits. The SVM-aware Jacobi2d improves the performance at $DOS = 109$ by more than 2X and improves the lower limit by 1.5X. SGEMM-svm-aware achieves a performance of 0.75 at $DOS = 156$, in comparison to near zero with the counterpart, resulting in a speedup by several orders of magnitude. We recognize this algorithm only scales to $DOS \approx 300$, and a different algorithm is needed if DOS grows past.

4.2 Driver Design Considerations

We focus on eviction-related implementation and design as eviction causes the most severe performance degradation. We first discuss implementations without changing the current design of policies and then discuss design alternatives.

Parallel Implementation. As shown in Figure 3, eviction is a synchronous event in the SVM driver, blocking migration of data

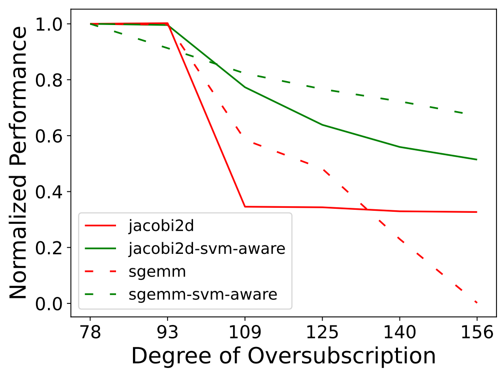


Figure 13: Comparison between the performance degradation of the original and SVM-aware Jacobi2d implementations.

needed in computation. Upon a migration request, SVM performs one eviction and then attempts the allocation again. In case of success, migration follows. Otherwise, SVM continues to perform another eviction and re-attempt until there is sufficient space. A migration involving one eviction results in doubled cost. A migration involving a series of evictions and allocation attempts not only increases the cost by multi-fold, but is error-prone. We observe in our experimental data that some allocations could take up to 15 seconds (a timeout in SVM for eviction; all these timeouts are removed in the figures).

Parallel implementations explore overlapping eviction and migration using multithreading, i.e., one thread per eviction/migration. Parallel implementations are feasible, as the driver has the knowledge of ranges to be evicted and migrated. Such parallelization, though involving locks, is expected to improve performance, especially for migrations involving multiple evictions. The current SVM already includes mechanisms for quick rollbacks in case of allocation failures. It can be adapted to support multithreading.

Eviction Policy. The Least Recently Faulted (LRF) policy evicts the range that has been migrated to the GPU earliest, ignorant of whether the range has been actively used or will be needed in the future while residing on the GPU. Consequently, it tends to evict the most intensely reused "hot" data from the GPU for SGEMM and alike, drastically degrading their performances.

A commonly used policy is the Least Recently Used (LRU). However, LRU requires the driver to timestamp page access on the GPU, which is too costly. Instead, we can explore simplified versions such as the Clock algorithm [20], which groups the ranges into two types: hot and cold. To avoid the prohibitive communication overhead between the device and the driver, the device could keep a copy of the range metadata and make the eviction decision. It would be trivial for the device to communicate the decision to the driver using existing communications.

Granularity. The large range sizes exacerbate eviction and the cost. A single data access missed on the GPU triggers the migration of the entire range. *Such behavior is the extreme case of aggressive prefetching*, which rapidly fills the GPU memory and leads to oversubscription and frequent thrashing.

Reducing the migration granularity or enabling adaptive "prefetching" would significantly benefit applications in Category III. Adjusting the range size would benefit applications with sparse or non-linear accesses and improve application SVM and alike. Instead of immediately migrating an entire range after one data request, the driver could migrate the range only after an access count has been reached or a percentage of data has been requested, similar to access counter based and density-based prefetching in UVM [15].

Zero-Copy instead of Demand Paging. The SVM driver supports both demand paging and zero-copy for accessing unified memory. With zero-copy, the device accesses data residing on the host at the cache line granularity. Each zero-copy incurs a large latency across the interconnect in comparison to local memory accesses. However, zero-copy is expected to benefit applications that experience severe thrashing under demand paging [35]. For such applications, the driver can allocate a portion of data in GPU memory for optimal utilization and allocate the remaining data on the host, accessing them via zero-copy.

5 RELATED WORK

Prior works studying the performance of unified memory systems can be looked at in two categories: (1) application-level analysis that focuses on the performance of select GPU applications utilizing different memory configurations and (2) system-level analysis that focuses on breaking down the components and features of unified memory implementations then directly profiling their cost. Most prior work exists in UVM with some work beginning to break into various HMM implementations, including SVM.

Application-Level Analysis works focus on the performance of GPU applications in unified memory models. These works span comparing performance between unified memory and programmer-managed GPU memory, analyzing the impact of various hardware on unified memory, identifying key application features impacting unified memory, and attempting to mitigate the performance loss of unified memory. Several works exist examining the performance differences between non-unified and unified memory across CUDA applications with combinations of prefetching & oversubscription [25, 26, 36, 46]. Chien et al. further examine the impact of memory hints and advice the programmer can provide in CUDA on application performance [11]. Xu et al. introduce a framework that relates application features with UVM hints, removing the need for programmers to decide which hints to use [45]. Gayatri et al. study the impact of Address Translation Services (ATS) for Power systems on unified memory [14]. All works discussed so far are in NVIDIA's ecosystem. To our knowledge, little work has examined unified memory in AMD's ecosystem except [21], which evaluates the performance of AMD GPU applications across user-managed memory, zero-copy remote access, and non-xnack managed memory.

System-Level Analysis works are the closest in relation to our work. Kim et al. identify the fault batching behavior of UVM, provide initial results on the relationship between batch size and batch cost, and propose a thread oversubscription technique for the GPU to mitigate large amounts of small batches [24]. Allen and Ge explore the driver-level impact of prefetching in UVM, provide initial insight on distinct components of UVM batches, and highlight how application access patterns influence GPU performance [1]. Allen and Ge further explore the components associated with servicing on-demand faults in UVM and how batch features influence the cost of various components [2]. Our work explores the behavioral features of AMD's unified memory, identifies the unique interplay between vendor UM and HMM, and analyzes the cost distribution in servicing on-demand page faults in an HMM-based unified memory.

Optimization works implement novel changes to unified memory drivers in order to lessen their performance cost. Chang et al. present an adaptive page migration scheme that enhances NVIDIA's UVM driver to dynamically migrate pages for irregular applications [10]. Ganguly et al. combine hardware prefetching and a novel pre-eviction policy to improve performance of oversubscribed UVM applications [13]. Li et al. propose a framework with proactive eviction, memory-aware throttling, and capacity compression to improve the performance of GPU oversubscription [31]. Such work is largely driven by application-level insights, while ours delves into the driver-level design and can be used in combination to optimize performance further.

6 CONCLUSION AND FUTURE WORK

In this work, we have examined the SVM design and its interface with the Linux kernel's HMM, and studied the performance impacts on a diverse set of GPU workloads. We find that the SVM design, especially the SVM ranges, are different from UVM. This design results in significant performance degradation for certain workloads under oversubscription. We show this performance degradation can be mitigated in part with SVM aware algorithms. We further discuss SVM design changes that would reduce the performance issues.

There is ample work to be done for SVM and in the larger field of unified memory. We have intentionally focused on demand paging from host to device, dismissing the directions from device to device and from device to host. The experiments presented throughout this paper solely use memory allocated through `hipMallocManaged()`. SVM/HMM allows other memory allocation such as `malloc()` and pinning on the host. Some combinations of various allocations may have observable performance effects.

Extending the work to different systems is another possibility. AMD's next generation of HPC processors, MI300 [5], are in the form of Accelerated Processing Units (APU) with tightly coupled CPU cores and GCDs in a shared processing unit. All CPU cores and GCDs share the same physical memory. The implications this architecture will have on SVM are unknown. Further analyzing HMM and its interface supporting various GPU devices is another avenue. At the start of this work, SVM was the only robust unified memory system utilizing Linux's HMM component. UVM has since been extended to support HMM [18].

Lastly, this work serves as a first step in analyzing and improving the current SVM implementation. The SVM-aware algorithm designs can serve as a base for other applications to improve performance under SVM. The discussed driver-level changes can be attempted and tested on actual systems.

ACKNOWLEDGMENTS

This research is partially supported by U.S. National Science Foundation under Grants CCF-1942182. This work was produced under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

[1] 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)*. 141–150.

[2] 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages.

[3] AMD. 2023. AMD ROCm™ documentation. https://rocm.docs.amd.com/en/latest/

[4] AMD. 2023. rocBLAS Documentation. https://rocblas.readthedocs.io/en/master/index.html

[5] AMD. 2024. AMD Instinct™ MI300 Series Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi300.html

[6] David A. Beckingsale, Jason Burmark, Rich Hornung, et al. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81.

[7] Tal Ben-Nun, Linus Groner, Florian Deconinck, et al. 2022. Productive performance engineering for weather and climate modeling with Python. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 73, 14 pages.

[8] Tom Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos. *J. Parallel Distrib. Comput.* 74, 12 (Dec. 2014), 3202–3216.

[10] 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* (Haifa, Israel) *(SYSTOR '21)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages.

[11] Steven Chien, Ivy Peng, and Stefano Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 50–57.

[12] Jake Choi, Heon Young Yeom, and Yoonhee Kim. 2022. Improving Oversubscribed GPU Memory Performance in the PyTorch Framework. *Cluster Computing* 26 (2022), 2835 – 2850. https://api.semanticscholar.org/CorpusID:253492267

[13] 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* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 224–235.

[14] 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)*. 41–46.

[15] Yanxiang He, Shaohua Wan, Naixue Xiong, and Jong Hyuk Park. 2008. A New Prefetching Strategy Based on Access Density in Linux. In *International Symposium on Computer Science and its Applications*. 22–27.

[16] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, et al. 2005. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.* 31, 3 (Sept. 2005), 397–423.

[17] Richard D. Hornung and Holger E. Hones. 2017. RAJA Performance Suite. [Computer Software] https://doi.org/10.11578/dc.20201001.36. https://doi.org/10.11578/dc.20201001.36

[18] John Hubbard, Gonzalo Brito, Chirayu Garg, et al. 2023. Simplifying GPU application development with heterogeneous memory management. https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/

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

[20] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA.

[21] Zheming Jin and Jeffrey S. Vetter. 2022. Evaluating Unified Memory Performance in HIP. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 562–568.

[22] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 207–221.

[23] Khronos Group. 2023. Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/api/opencl

[24] Hyojong Kim, Jaewoong Sim, Prasun Gera, et al. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1357–1370.

[25] 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. *The Journal of Supercomputing* 75, 11 (Nov. 2019), 7625–7645.

[26] 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)*. 1–6.

[27] Lawrence Livermore National Lab. 2022. Tioga. https://hpc.llnl.gov/hardware/compute-platforms/tioga

[28] Lawrence Livermore National Lab. 2023. El Capitan: Preparing for NNSA's first exascale machine. https://asc.llnl.gov/exascale/el-capitan

[29] Teven Le Scao, Angela Fan, Christopher Akiki, et al. 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv:2211.05100 [cs.CL]

[30] Edgar A. León, Trent D'Hooge, Nathan Hanford, et al. 2020. TOSS-2020: A Commodity Software Stack for HPC. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 40, 15 pages.

[31] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, et al. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 49–63.

[32] Linux Kernel Development Community. 2023. Heterogeneous Memory Management (HMM). https://www.kernel.org/doc/html/latest/mm/hmm.html

[33] Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, et al. 2023. A Research Retrospective on AMD's Exascale Computing Journey. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages.

[34] Seungwon Min, Kun Wu, Sitao Huang, et al. 2021. PyTorch-Direct: Enabling GPU Centric Data Access for Very Large Graph Neural Network Training with Irregular Accesses. *CoRR* abs/2101.07956 (2021). arXiv:2101.07956

[35] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, et al. 2020. EMOGI: Efficient Memory-Access for out-of-Memory Graph-Traversal in GPUs. *Proc. VLDB Endow.* 14, 2 (oct 2020), 114–127.

[36] 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 Transactions on Parallel and Distributed Systems* 27, 6 (2016), 1579–1588.

[37] Oak Ridge National Lab. 2022. Frontier User Guide - OLCF User Documentation. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

[38] Minh Pham, Yicheng Tu, and Xiaoyi Lv. 2023. Accelerating BWA-MEM Read Mapping on GPUs. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 155–166.

[39] Vara Prasad, William Cohen, FC Eigler, et al. 2005. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*. New York, NY: IEEE, 49–64.

[40] Alan Smith and Norman James. 2022. AMD Instinct MI200 Series Accelerator and Node Architectures. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 1–23.

[41] Top500. 2023. November 2023. https://top500.org/lists/top500/2023/11/

[42] Unified Acceleration Foundation. 2023. oneAPI. https://www.oneapi.io/spec/

[43] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[44] Martin Winter, Daniel Mlakar, Rhaleb Zayer, et al. 2018. faimGraph: High Performance Management of Fully-Dynamic Graphs Under Tight Memory Constraints on the GPU. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 754–766. https://doi.org/10.1109/SC.2018.00063

[45] Hailu Xu, Pei-Hung Lin, Murali Emani, et al. 2022. XUnified: A Framework for Guiding Optimal Use of GPU Unified Memory. *IEEE Access* 10 (2022), 82614–82625.

[46] Qi Yu, Bruce Childers, Libo Huang, et al. 2019. A quantitative evaluation of unified memory in GPUs. *The Journal of Supercomputing* 76, 4 (nov 2019), 2958–2985.