Holistic Performance Analysis and Optimization of Unified Virtual Memory

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Tyler Allen
May 2022

Accepted by:
Dr. Rong Ge, Committee Chair
Dr. Amy Apon
Dr. Jon Calhoun
Dr. Jacob Sorber

The programming difficulty of creating GPU-accelerated high performance computing (HPC) codes has been greatly reduced by the advent of Unified Memory technologies that abstract the management of physical memory away from the developer. However, these systems incur substantial overhead that tends to paradoxically worsen for codes where these technologies are most useful. While these technologies are increasingly adopted for use in modern HPC frameworks and applications, the induced performance cost reduces the efficiency of these systems or turns away some developers from adoption entirely at the cost of significantly more complex codebases. In this work, we take the first deep dive into a functional implementation of a Unified Memory system to evaluate the performance and characteristics of these systems. We focus on an NVIDIA UVM-based system and investigate the root causes of the overhead to the UM system. The costs in this system are complex and difficult to analyze due to the long list of hardware and software constituents and the requirement of targeted systems-level analysis methods. In this thesis, we investigate the architecture of a Unified Memory system and reveal the internal behaviors of page fault generation and servicing. We show specific GPU hardware limitations using targeted benchmarks to uncover driver functionality as a real-time system when processing the resultant workload. We further provide a quantitative evaluation of fault handling for various applications under different scenarios, including prefetching and oversubscription. We find that the driver workload is dependent on the interactions among application access patterns, GPU hardware constraints, and Host OS components. We determine that the cost of host OS components is significant and present across UM implementations. We also provide a proof-of-concept asynchronous approach to memory management in UVM that allows for reduced system overhead and improved application performance. This study serves as a proxy for future shared memory systems and provides constructive insight into future implementations and systems, such as those that interface with future Heterogeneous Memory Management technologies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graphics Processing Units (GPUs) have become one of the computational devices in modern high-performance computing (HPC) systems. As of November 2020, six of the top ten Supercomputers on the Top500 are NVIDIA GPU-Based systems[1], and 67% of the Top500 in total as of June 2020[20]. GPUs are now relied upon to enable many modern scientific and machine learning-based applications at critical research centers around the world.

GPUs are a type of *hardware accelerator* - a specialized hardware unit that functions alongside the CPU in a typical system. Due to the success of GPUs, other types of accelerators are gaining recognition in HPC systems, including Data Processing Units (DPUs) and Field-Programmable Gate Arrays (FPGAs). In HPC systems, accelerators are usually present as dedicated add-on cards connected to the main CPU over some high-speed interconnect. Accelerators typically have their own memory hierarchies and programming models. The inclusion of these accelerators has greatly increased the management and programming complexity of these systems has risen alongside the performance.

As the inclusion of GPUs and other accelerators has and continues to improve the theoretical performance of HPC systems, the complexity of programming for high-performance increases two-fold. First, efficient utilization of accelerators requires effective programming against the target architecture. Adapting to the programming models and paradigms of these devices is difficult even for professional programmers, but HPC systems are often programmed by domain scientists who may not have formal training, increasing the difficulty substantially. Second, the increase throughput enables more advanced domain applications that were previously intractable on lower-throughput

systems. The applications themselves increase in complexity, placing an even greater burden on programmers when paired with an increasingly complex programming environment. Due to this, the barriers to creating and maintaining correct, high-performance applications for advanced domain problems have been steadily rising.

A large step towards reducing the complexity introduced by increasingly heterogeneous systems has been the introduction of **heterogeneous shared memory** systems. These systems manage separate host and device physical memory domains in systems software, giving the programmer a transparent view of memory and removing the need for programmers to manage these domains themselves. *Heterogeneous Memory Management* (HMM) and NVIDIA Unified Virtual Memory (UVM) are two independent yet collaborative heterogeneous shared memory implementations. HMM is an upcoming Linux kernel feature providing a generic front-end for OS memory management for commodity systems, but primarily only serves as a host-side virtual memory manager and as a common front-end for device-specific drivers [12, 23]. In contrast, UVM is an all-in-one approach to heterogeneous shared memory that provides near-transparent memory management support for NVIDIA devices only. It can also serve as a backend to interface with HMM [12, 23], although mutual support is still in development [39]. Because HMM is still under development, NVIDIA UVM is currently the most represented heterogeneous shared memory technology in HPC and has been adopted into several common HPC frameworks such as Raja [6], Kokkos [7], and Trilinos [22].

These technologies require hardware and software support to effectively implement the underlying memory management. They create a virtual shared memory space between the host and device by extending the host's virtual **demand paging** system. With appropriate hardware support, modern NVIDIA GPUs support the full Linux virtual memory paging scheme, particularly the capability of generating **page faults** when memory is not mapped into GPU page tables. In the event of a page fault, GPU hardware will issue an interrupt to the host CPU. These interrupts are handled by kernel-mode drivers on the host, which will result in the migration of the needed data to the GPU as well as appropriate updates to GPU page tables. This process is complex, and represents a clear source of performance sensitivity as it acts as a data bottleneck in the event of page faults.

The ease of use provided by these technologies comes at the cost of significant overhead. Prior work has shown that many HPC applications suffer significant performance losses from UVM in many HPC applications and micro-benchmarks [19, 17, 26, 11, 17]. This trade-off varies from

high to unacceptable depending on the application. However, the root sources of these costs are not well studied. While some work has taken steps to improve performance, they generally address two features within UVM, runtime-prefetching and page eviction, without addressing the root causes; additionally, most of these approaches require hardware changes and are therefore not applicable to existing systems [43, 16, 15, 24].

Key Problem: The performance gap between heterogeneous shared memory and directly-managed memory in modern systems must be fully understood and then resolved to enable effective utilization of increasingly complex HPC without shifting the responsibility of managing complexity onto domain scientists. The emergence of these technologies is of great importance to maintaining a manageable level of programming complexity for HPC systems, but the performance gap will continue to relegate their utility to the prototyping stage for many applications if these issues are not addressed. While prior work has taken steps to understand the performance of these systems, the underlying performance issues are not well understood and therefore challenging to resolve. In this work, we seek to fully understand the performance features of these systems and resolve key underlying issues for existing and future systems.

This thesis seeks to address the performance gap between directly-managed memory and heterogeneous shared memory in support of high-performance heterogeneous shared memory systems using the following approach: (1) use UVM as a test-bed technology for thoroughly to identify and quantify the root costs of these technologies from a systems-software perspective, (2) evaluate which costs are UVM-specific vs. generic costs that would apply to all devices with HMM to support future implementing devices, and (3) uses these findings to mitigate the identified host OS latencies that contribute a significant portion of overhead to UVM. We utilize device-instrumented timers and software performance counters to identify key areas of performance loss. By tracing the root cost to hardware, UVM driver, and host OS functionality, we identify targeted optimizations that improve the overall performance of any application utilizing UVM. Additionally, we provide design insights for future HMM-implementing hardware and software.

1.1 Demand Paging Virtual Memory

Virtual Memory is an Operating System (OS) abstraction over physical hardware memory that enables memory to act as a cache for other hardware locations [36]. Specifically, it allows user

programs to refer to memory that may not be present in physical memory, but is located on a different hardware than main CPU memory. As with any cache, if the backing hardware is larger, then user programs are able to operate with more data than there is space in the cache, physical memory, known as **oversubscription**.

In order to implement virtual memory, most systems today use **demand paging**. Demand paging is a memory management scheme that splits memory into equal, fixed-size pages and **migrates** them between hardware locations when accessed, otherwise known as *swapping*. Instead of tracking at the byte level, pages typically represent at least several thousand sequential bytes that are tracked using data structures known as **page tables**. Accessing a virtual memory address when the data is not present in physical host memory is a virtual memory miss, typically referred to as a **page fault**. This scheme usually requires hardware and software support. There are two key benefits from demand paging in addition to those provided from virtual memory: (1) pages allow reduced tracking and management overhead , and (2) a guaranteed amount of bulk data transfer during page faults.

1.2 Heterogeneous Shared Memory

Hardware accelerators open the door to all new use cases for virtual memory. While virtual memory with demand paging was created to allow for bulk storage devices to back host memory (and greatly simplify multi-user systems), these same ideas can be applied to any discrete memory or storage device. GPUs and other add-on accelerators typically have their own discrete, non-expandable memory that is relatively small in relationship to host memory on HPC systems, mirroring the physical memory/bulk storage scenario in many ways. Accelerators could issue page faults to the CPU when memory is required and having the host OS fulfill them. In practice, several systems exist to implement this kind of technology, which we refer to as **heterogeneous shared memory**.

A key difference between traditional paging virtual memory systems backed with storage and accelerators is the importance of system overhead. In legacy storage devices, the hardware itself is orders of magnitude more expensive than any software operations. In contrast, host and accelerator memories are both fast, and HPC systems also have high performance requirements. This creates a scenario where systems software itself is the key bottleneck. For many workloads, this bottleneck contributes unacceptable overhead and limits the utility and adoption of heterogeneous

memory systems.

1.2.1 Explicitly Managed Memory

The de facto programming model for heterogeneous accelerators is through explicit programmer management. User programs need to explicitly allocate host memory, as usual, in addition to device memory. For data to be used on a given device, the programmer must explicitly manage its location and migrate it between devices as needed. Listing 1.1 demonstrates this using the CUDA runtime framework for NVIDIA GPUs. On lines 3-4 by allocating host memory into variable `host_a` and then initializing it. On line 5, we allocate device memory for line `device_a`, and then on line 6 we copy the data from `host_a` in host memory to `device_b` in device memory. After this, the device memory is ready to be used by the GPU for computation on line 7. Finally, the memory must be copied back to the CPU (line 8), and then the runtime must be synchronized (line 9) to ensure that the GPU operation queue is empty before accessing the data on the CPU again. This is a complex process for a fairly simple example. More complex data structures with dynamic layouts and indirect pointers complicate this further. However, explicit management presently offers the best overall performance for effectively programmed applications.

Listing 1.1: A demonstration of direct memory management, requiring explicit allocation and data migration both to and from the GPU during execution.

```
1  —— snip ——
2      float* host_a, *device_a;
3      host_a = (float*) malloc(ARRAY_SIZE);
4      memset(host_a, ARRAY_CONST, ARRAY_ENTRIES);
5      cudaMalloc(&device_a, ARRAY_SIZE);
6      cudaMemcpy(device_a, host_a, ARRAY_SIZE, cudaMemcpyHostToDevice);
7      example_kernel<<<GRID_DIM, BLOCK_DIM>>>(device_a);
8      cudaMemcpy(host_a, device_a, ARRAY_SIZE, cudaMemcpyDeviceToHost);
9      cudaDeviceSynchronize();
10 —— snip ——
```

1.2.2 Unified Virtual Memory

Unified Virtual Memory (UVM) is NVIDIA's approach to heterogeneous shared memory using demand paging. UVM is implemented as a kernel driver module in Linux. This module extends the Linux paging system and operates with the proprietary NVIDIA GPU driver for GPU memory management. UVM is not fully transparent and requires special memory allocators through the CUDA runtime to receive memory that is managed by UVM. The UVM driver operates as the focal point of the system, receiving page faults and necessary correct operations to other system components.

UVM is frequently discussed alongside two main features: prefetching and evictions. Prefetching is a UVM driver optimization intended to speed up overall UVM processing. This feature is enabled by default. Eviction is a feature that enables oversubscription in UVM, allowing UVM to swap pages between devices when device memory is full.

We can see the same example from listing 1.1 implemented in UVM in listing 1.2. The scenario is greatly simplified, requiring only a single pointer for both host and device. The special `cudaMallocManaged` function takes the place of `malloc`, `cudaMalloc`, and `cudaMemcpy` through the implementation of demand paged virtual memory. This greatly simplifies the programming requirements, but support is limited to NVIDIA GPUs. UVM supports a range of functionalities in addition to the paged migration functionality, but require additional programmer support. We focus on paged migration as the primary UVM functionality and best reflecting the demand paged virtual memory functionality.

Listing 1.2: A simple example corresponding to listing 1.1, demonstrating the reduction in code complexity provided by UVM.

```
1 -- snip --
2     float* a;
3     cudaMallocManaged(&a, ARRAY_SIZE);
4     memset(a, ARRAY_CONST, ARRAY_ENTRIES);
5     example_kernel<<<GRID_DIM, BLOCK_DIM>>>(a);
6     cudaDeviceSynchronize();
7 -- snip --
```

1.2.3 Heterogeneous Memory Management

Heterogeneous Memory Management (HMM) represents the ideal experience for programmers. HMM is a Linux kernel feature for implementing demand paging virtual memory for arbitrary devices. In example 1.3, we can see that the system allocators take the place of special CUDA allocators, allowing programmers to only focus on device code and ignore memory management. This represent the ideal memory management scenario for programmers.

Listing 1.3: An example of HMM, allowing fully transparent memory management by the system for all GPUs on the system.

```
1 ---- snip ----
2       float* a;
3       a = malloc(ARRAY_SIZE);
4       memset(a, ARRAY_CONST, ARRAY_ENTRIES);
5       example_kernel<<<GRID_DIM, BLOCK_DIM>>>(a);
6       cudaDeviceSynchronize();
7 ---- snip ----
```

1.2.4 UVM as a Case Study

Despite the ideal programmer circumstances provided by HMM, we focus on UVM to demonstrate performance constraints for heterogeneous shared memory. This is because HMM is presently not usable in the general case, even for NVIDIA GPUs, due to ongoing development. However, HMM is only able to provide a front-end and host-side data management. HMM requires vendor drivers to register themselves with HMM, and provide device-management support. This means that HMM will represent a common interface and limited host page support for devices, but the majority of work will be performed by device-specific drivers. In practice, UVM will eventually act as the NVIDIA-specific backend for HMM, and presently can do so in limited cases. In this sense, UVM is highly representative of how HMM will perform after its full development for NVIDIA GPUs. In this work, we recognize that HMM is likely an inevitable standard, and so our findings on UVM to draw out insights or improvements for the final HMM, as well as other driver implementations, implementation where applicable.

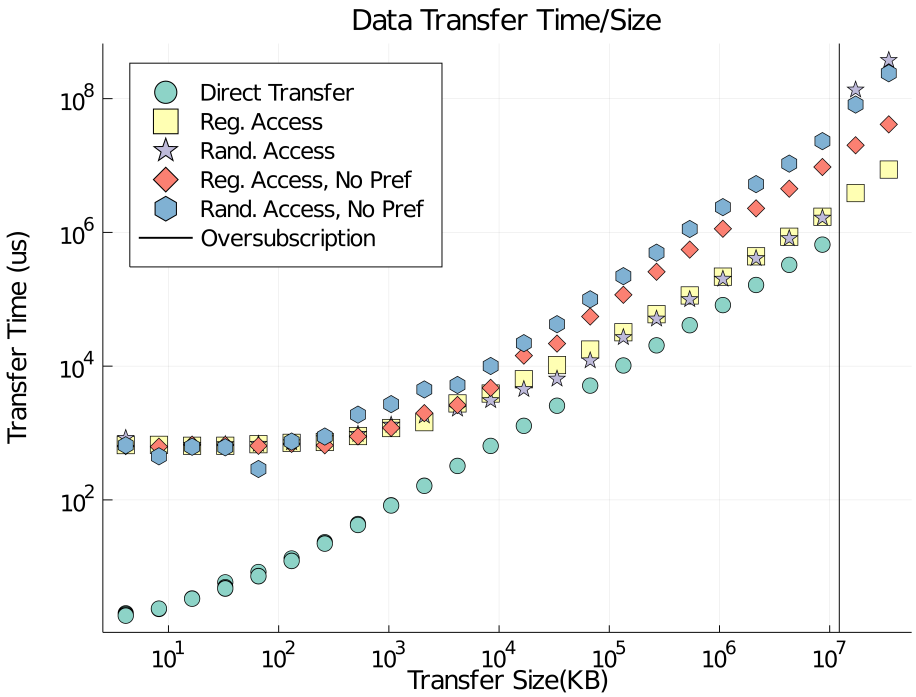


Figure 1.1: UVM access incurs one or more orders of magnitude greater latency than direct data transfer.

1.3 The Cost of UVM

UVM can have a significant negative impact on performance, as as has been shown by many prior works [19, 17, 26, 11, 17]. In figure 1.1, we use three benchmarks to demonstrate this cost: a direct bulk transfer example, a UVM benchmark that accesses a fixed number of pages using an extremely regular page-level access pattern, and a UVM benchmark that accesses a fixed number of pages in random order. We run these applications with prefetching both enabled and disabled, up to some problem sizes that are larger than GPU memory to create oversubscription. There are three key observations: (1) the latency of UVM access is an order of magnitude great than that of explicit direct management, (2) for in-core workloads, prefetching can narrow the performance gap but still have several times worse performance, and (3) once the GPU global memory is oversubscribed, the performance latency for the random access pattern increases by an order of magnitude. Ideally, UVM accesses are expected to incur the same latency as direct memory management regardless of access pattern or oversubscription. This motivates us to identify the root sources of performance gaps and identify solutions to these issues.

Locating and mitigating the performance gap is non-trivial for several reasons. First, UVM performance has primarily been studied at the surface level for programmer viability instead of the systems software level, leaving a lot of open questions about root causes. Second, the demand paged virtual memory architecture converts massively parallel data accesses into a mixed data stream for systems software to process sequentially, which greatly complicates the processing within UVM as well as functionality such as prediction and prefetching. Third, UVM is faced with the trade-off between data management complexity, e.g. page tables, and the constraints imposed by limited memory capacity. The performance bottleneck may have multiple sources and their dominance may vary with access scenarios. This is more challenging than on CPU systems where accesses to caches at a time are a cache block (e.g., 64 bytes) and main memory has a larger space in hundreds of Gigabytes or Terabytes.

1.4 Summary of Contributions

In this work, we seek to understand the root causes of performance in heterogeneous shared memory systems, resolve fundamental architectural issues that stem from systems software, and identify areas where hardware designers and proprietary software creators can make improvements or utilize UVM innovations for future iterations. Our contributions are as follows:

- A high-level overview of UVM performance and workload creation using software timers and counters within the UVM driver software. This provides an understanding of the most time-consuming components of software as well as an intuitive understanding of the UVM workflow. We find that UVM fault handling and GPU computation are generally synchronous and are not able to significantly overlap. We also find a general roadmap of the areas of the UVM software that need to be investigated further.

- A fault-batch-oriented analysis of UVM performance using detailed timing of low-level functionalities along the GPU fault path, allowing us to identify the key sources of cost within UVM. This allows us to locate key targets of optimization. It also allows us to isolate sources of overhead into their specific components as well as NVIDIA or GPU-specific, or common code that could impact many devices in the future HMM implementation. This enables targeted optimizations to specific areas, as well as a broader understanding of UVM ideas to future heterogeneous shared memory implementations.

- We extend the prior analyses to include prefetching and oversubscription workloads within UVM to gain a deeper understanding of real application scenarios and the impact of these features on the baseline UVM system by examining both batch-feature aspects and page-level access patterns for workloads with these features enabled. This provides the most comprehensive analysis of these features from a systems perspective thus far, demonstrating the effects of prefetching and oversubscription on the workload and exposes the most impactful components of overhead on real workloads.

- We use our findings to apply specific design changes to host-OS and HMM-impacting kernel and driver components where the general purpose of the host OS conflicts with the real-time requirements of HPC accelerators. This approach requires careful changes to ensure the continued operation of all system and driver components without inadvertently impacting performance of other components on the system, but is able to resolve a large percentage of overall UVM overhead with design implications for improving the performance of all implementing devices in HMM systems.

These key findings give us a high-level understanding of UVM performance and help outline the parts that are reusable in future implementations as well as optimize the components that we were able to perform.

Chapter 2

Background

In this chapter, we offer an overview of UVM architecture, as well as details on driver functionality and organization. All references to GPU architecture are based on NVIDIA GPUs specifically. We also contrast this implementation with the design of HMM.

2.1 GPU Architecture

GPUs are highly-parallel accelerators that have several commonalities with traditional computing architectures, but also key differences that are important to conceptualizing how technologies like demand paging function in practice on these devices.

2.1.1 Programming Model

CUDA code is executed in code blocks known as **CUDA kernels**. CUDA kernels are written very similarly to traditional C functions, with special designations to indicate it should be compiled as device code. When CUDA kernels are called (launched) in C programs, there are a few key differences to traditional functions. First, they are executed asynchronously, and the CPU thread continues to execute after the kernel launch. Second, kernel launches create many threads executing the same code block simultaneously.

When a kernel is launched, all threads within that kernel are referred to as the kernel **grid**. The kernel is complete when all threads within the grid have returned. Threads in the kernel are grouped into schedulable entities called **thread blocks**. All threads within the kernel execute the

same function with the same parameters. However, each block has a unique ID value to distinguish it from other blocks, and a unique thread ID within that block. Using these two ID values, each thread is uniquely identifiable, and can use this distinction in its computation. The programmer decides the size of thread blocks, either statically or dynamically computed at runtime. Threads within a block will be available to be scheduled at a given time. The main programming considerations with thread blocks are (1) shared data and data locality between warps, as they will be scheduled together, and (2) synchronization requirements, as thread blocks are the largest unit that is easily synchronized.

User threads within a kernel are divided into **warps**. A warp is the minimal schedulable unit within CUDA, which is generally 32 threads. CUDA uses a same-instruction-multiple-thread (SIMT) execution model, allowing threads within a warp to execute the same instruction in parallel. If threads need to execute different instructions, they are *warp divergent*. Divergent threads cannot execute simultaneously, and instead must be serialized. This programming model gives the programmer the responsibility of designing their algorithms to ensure warps are executing the same instruction wherever possible.

2.1.2 Simultaneous Multiprocessors

For NVIDIA GPUs, thread blocks are scheduled to "Streaming Multiprocessors" (SMs). There are many SMs per GPU, up to 128 on the A100 architecture [33]. Each SM contains many arithmetic units referred to as **CUDA cores**, which can perform floating point or integer operations. Alongside CUDA cores are memory load/store units, special function units (SFUs) for canned expensive operations such as modulus, and in newer architectures special half-precision tensor cores for even greater parallelism. Additionally, each SM has a local "unified cache," which generally functions as an L1 cache[1], as well as a register file for block contexts.

Within a single SM, resources are evenly partitioned between one or more *warp schedulers*. Each SM scheduler contains a fixed number of CUDA cores, 16 on newer architecture, and a proportional number of the other functional units. As the name implies, warps from one-or-more thread blocks are chosen by the warp scheduler to execute on a given cycle. While warps are stalled, such as for a memory access, other warps can be scheduled to execute. The per-SM register file is severely over-provisioned relative to the number of execution units, allowing many more warps to have local

[1]The L1 cache is inclusive of shared memory and texture caches that are not material to our later discussion.

contexts than the number of warp schedulers available; this is explicitly for the purpose of hiding latency.

2.1.3 Memory Hierarchy

The overall memory hierarchy on NVIDIA GPUs is very similar to some modern CPUs; each SM has a local L1 cache, all SMs share an L2 cache, and finally there is a segment of addressable graphics (GDDR) or high-bandwidth memory (HBM). SMs are grouped into "texture processing clusters" (TPCs), including two SMs on the Volta architecture; SMs in the same TPC share a micro translation look-aside buffer (µTLB). On an address translation miss, such as those that may result in a page fault, the µTLB may pause the relevant warp schedule to prevent continued fault generation until it is resolved [45].

Page faults on user data are specifically implemented as **replayable faults** in NVIDIA GPUs. Replayable faults do not block the faulting GPU compute unit, which can continue running non-faulting warps until a `replay` command is received[45]. However, these faults can still lead to data dependency stalls when the data is required, and an interrupt is sent to the host. Once the page fault is fulfilled, the host will issue a **fault replay**. During a fault replay, the stalled SMs will re-execute instructions that generated a page fault. If the fault was satisfied then execution will continue as normal, otherwise the instruction will fault again.

2.1.4 Hardware Interconnect

The add-on accelerator card format requires some form of local hardware interconnect between the accelerator and the host CPU. Generally, these interconnects connect all add-on cards to each other and the host CPU, but must pass through the host CPU for access to other hardware resource such as host memory. For example, access to memory can be streamlined through technologies like remote direct memory access (RDMA) after setup through the host CPU.

The two most common interconnects at the time of writing are PCIe (peripheral component interconnect express), supporting most available accelerators, and the proprietary NVLINK2 interconnect which supports some NVIDIA products on Power9 architectures. Due to the proprietary nature of NVLINK2, these interconnects can both appear on a single system; NVLINK2 may connect all GPUs, with a PCIe switch from GPUs to the main CPU on an x86 system. The primary features

of PCIe against its predecessors are relatively low latency and relatively higher bandwidth with PCIe 4.0 on a standard x16-width bus offers nearly 32GB/s throughput. In contrast, NVLINK2 is a more modern interconnect, offering up to 50GB/s per link, with the V100 GPU having 6 links for a total of 300GB/s [34]. The key difference in interconnect features for UVM, aside from bandwidth, is that NVLINK2 is capable of coordinating with a Power9 architectural feature, Address Translation Service (ATS), to provide hardware-accelerated address translation[40]. Prior work has shown that this feature can speed up UVM to some extent [17]. Further, the current implementation of HMM only functions with Power9 ATS and NVLINK2 [17, 23].

Experiments in this work primarily concern PCIe connections. The rationale is that the hardware may influence the cost of individual operations, but the fundamental functionality of UVM is not greatly affected by hardware. The one exception is ATS, where much of CPU-side page translation has largely been shifted out of software; however, it is not clear that ATS will ever be supported or available to other architectures and accelerators. Due to this and resource limitations we focus our work on PCIe, although in future work we will investigate other architectures as they become available.

2.2 UVM Architecture

The UVM architecture is abstractly a client-server architecture between one or more software clients (user-level GPU or host code), and the server (open-source `nvidia-uvm` driver) servicing page faults for all clients. In this section, we overview the overall UVM architecture, the delegation of functionality within the architecture in response to page faults, and the overall communication patterns.

2.2.1 Memory Access Types

UVM supports three page access behaviors:

- **Paged migration** moves data between devices in response to a page fault, maps the page into the faulting device's physical space, and unmaps from the previous location.
- *Remote Mapping* maps the requested data into the requester's page tables without actually migrating it and accesses it using DMA or a related mechanism, and

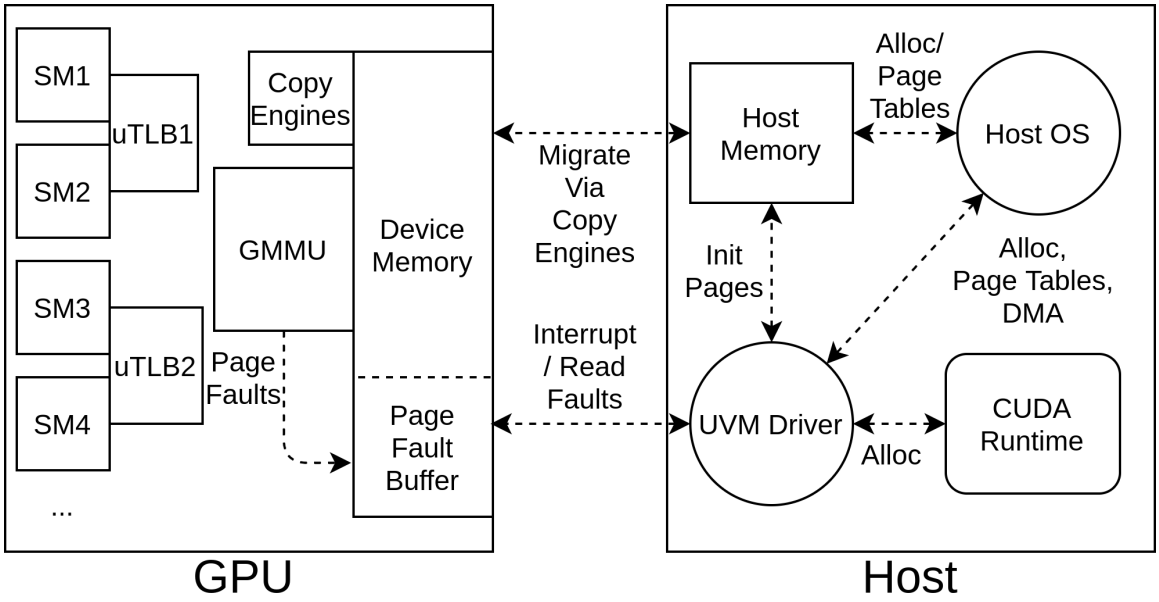


Figure 2.1: Faults propagate from SMs through the µTLB to the GMMU, which can write faults directly to the fault buffer. The UVM driver reads groups or *fault batch* of faults from the fault buffer. After processed, the requested data is migrated to the GPU through copy engines and page mapped appropriately.

- *Read-only duplication* duplicates data at two or more physical devices and maps them locally to each device under the constraint that the data cannot be mutated. We focus on no oversubscription where application data fits in the GPU global memory in this section and discuss oversubscription in detail later.

Our work focuses specifically on paged migration, as it is the default and primary behavior for UVM and the functionality HMM expects device drivers to perform. The other two situations arise when programmers explicitly requested them by assigning attributes to memory allocations using *memory access hints*.

2.2.2 Memory Management and Communication

Management Delegation. As part of servicing page faults, the UVM driver acts as an intermediary management layer that integrates NVIDIA GPUs into the host OS memory system with dependencies on the main proprietary `nvidia` driver/resource manager and the host OS for memory management. UVM tracks metadata for all memory locations, including which devices have mappings to memory locations and the physical location(s) of memory. For memory operations involving host memory, UVM utilizes the host OS programming interface to handle physical allocations and

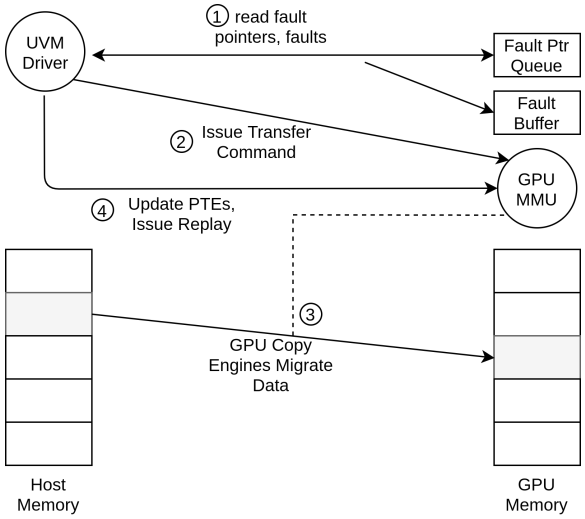


Figure 2.2: UVM Fault-Handling Communication Architecture. (1) The UVM driver reads fault pointers out of a queue. The pointers lead to the specific fault entry in the fault buffer. Faults are cached on the host. (2) After processing, the driver will alert the GPU of data to be transferred. (3) The GPU will initiate the data transfer to GPU memory through DMA. (4) After one or more repetitions, the CPU will issue a replay to the GPU. Omitted: The UVM driver is initially prompted by a GPU-originating trap.

the host's page tables. For GPU memory, UVM manages the page tables and physical memory location, but relies on the `nvidia` driver as the physical memory allocator.

Handling Page Faults. The UVM driver services all page faulting interrupts for memory allocated as CUDA managed memory. All GPUs and host processes can function as clients for the UVM driver; the high-level host-device communication relationship is illustrated in figure 2.1. As with typical host allocations, CUDA managed memory is first allocated virtually but not physically. On *first touch*, a page fault will be generated by the CPU or GPU seeking access to the data. The UVM driver will receive these page faults, regardless of the device of origin, and will allocate physical memory on the device where the data is accessed. For later accesses, when the physical memory exists but a page fault is generated by a different device, the UVM driver will instead **migrate** the data to the faulting device from its original location. Paged migration between host and device, or vice versa, requires coordinating both drivers to create direct memory access (DMA) mappings, copy data, and handle virtual memory mapping/unmapping for both devices.

Figure 2.2 is a more detailed depiction of the driver's actions in responding to a page fault. Faults are delivered in a two-part message to the host. First, the host UVM driver is alerted of a page fault via an interrupt over the interconnect. Second, the driver retrieves fault information required to service the fault and tracking information stored in the *fault buffer* on the GPU-side.

The data required to service the fault is stored in a separate GPU fault buffer, which the host will read to start the servicing operation. After internal processing and unmapping the requested pages from the host page tables, the host will instruct the GPU to copy the data from host memory to GPU physical memory. Afterwards, the CPU will update page tables and issue a **fault replay** instruction. The key difference for HMM is that the common host interrupt handler receives the fault and unmaps CPU pages if necessary, but will deliver it to the UVM driver; overall the effective differences are minor.

2.2.3 Internal UVM Design and Organization

Memory Organization. UVM defaults to the host OS page size, as it mainly operates as an extension to the existing virtual memory system. The default page size is 4KB for x86 systems, and 64KB for Power9 systems - the two most commonly used architectures with NVIDIA systems at the time of writing. Internally, UVM uses several layers of abstraction overtop of pages. UVM uses a four-level hierarchy for memory address space: address spaces, virtual address ranges, virtual address blocks, and pages. In general, a virtual address space is associated with a user process. Each address space is composed of "ranges", each corresponding to an arbitrarily sized memory allocation i.e. `cudaMallocManaged()` or related allocator. A range is broken up into 2MB sequential **virtual address blocks**, VABlocks. VABlocks are page-aligned and are composed of OS pages. Virtual address blocks serve as a logical boundary for management, data transfers, page table updates, and most other operations within the UVM driver.

Batching. In practice, UVM handles faults in **batches** as an optimization. Outstanding faults are grouped into batches and cached on the host during servicing. Batches default to a maximum size of 256 faults, although a batch will begin servicing with less faults when the fault buffer is exhausted. Batches potentially reduce the amount of unique data transfers, page table updates, and fault replays required. However, batching still respects the logical separation of VABlocks - faults within a batch are further sorted into their respective VABlock, and serviced in full. Operations such as data transfers and virtual memory updates are only coalesced within a single VABlock.

Algorithm. We coarsely categorize the operations of the UVM Driver in three groups: pre/post-processing, fault servicing, and fault replay policy. During *pre-processing*, the driver stores page fault information read from the GPU fault buffer and sorts them by address. Faults are fetched until all the fault pointer queue is empty, the current batch of faults is full, or fault that is not ready

is encountered, depending on policy. The driver will service each VABlock within the fault batch individually. Fault servicing involves prefetching, memory allocation, updating page tables, data transfer, and possibly issuing one or more fault replays or other operations, subject to the *fault replay policy*.

Fault Replay Policy. Fault replays are an expensive operation, requiring full GPU µTLB flushing, SM synchronization, and synchronization with the host. Additionally, any unfulfilled faults will issue a duplicate fault after a replay. Replaying more frequently increases total overhead and the number of duplicate faults. However, replaying infrequently increases the amount of time that SMs are starved for data on the GPU. The implemented replay policies are:

- **Block Policy**: A replay is issued when faults for a single VABlock within a batch have been serviced. This policy issues the replay notification the earliest and the most frequent. Per the driver, this policy allows for faulting SMs to resume earlier at the cost of more replays.
- **Batch Policy**: A replay is issued after each full fault batch has been serviced. This policy has less frequent replays, but incurs larger latency before SM execution can resume.
- **Batch Flush Policy**: This policy is the same as batch policy, but the fault buffer is flushed after a batch is completed but before it is replayed. The replay will cause all faulting warps to resume, even if the faults are not satisfied, forcing them to fault again. Flushing the buffer prior to this prevents duplicates from appearing in the buffer at the cost of remote queue management.
- **Once Policy**: A replay is issued after servicing as many batches as possible, leaving the fault buffer empty. This policy is the extreme case with minimal replays but long latency.

The default is the *batch flush policy*, as it is a compromise in the frequency of replays with additional mitigation for duplicate faults.

2.3 Prefetching and Eviction in UVM

2.3.1 Prefetching

The UVM driver implements a runtime prefetcher to reduce latency. Prefetching is implemented in two stages, and is invoked per-VABlock present in a batch. First, on x86 systems, every

4KB page is "upgraded" to the corresponding virtual 64KB aligned "big page." This prefetching stage is relatively simple but serves two main purposes. (1) This stage satisfies common cases of spatial locality by prefetching the local data around a faulted page and allows for larger bulk transfers. (2) This behavior emulates the behavior of Power9 systems (64KB pages) on x86 systems (4KB pages), allowing simplified reasoning for other sections of code while prefetching is enabled.

The second prefetching stage uses NVIDIA's implementation of a "density prefetcher" [21]. This prefetching mechanism is sometimes referred to as a tree-based prefetcher in other works [15]. Within the prefetcher, each VABlock is conceptually represented by a 9-level ($\log_2(\frac{2\text{MB}}{4\text{KB}})$) binary tree. The leaves of the tree represent sequential 4KB pages within the VABlock. Higher nodes represent the *subtree access density*, e.g. the number of leaves in the subtree that are already located on the GPU *or* are present in the fault batch including pages flagged for prefetching by the upgrade to "big pages". For each of the nine levels starting from the leaf representing the faulted page, the prefetch region is defined as the largest subtree such that the *subtree access density* exceeds the *density threshold*. The density threshold is a $1-100$ value set at driver load time with a default of 51. Therefore, by default, if more than 51% of leaves in a subtree are resident on the GPU or are presently faulted, the entire subtree will be fetched.

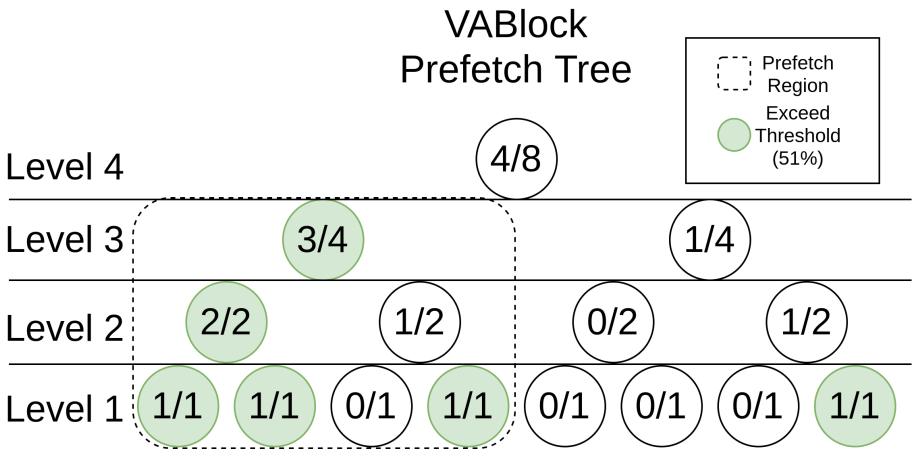


Figure 2.3: Illustration of the current UVM page prefetching tree concept. The access density threshold is 51%, the default value. Leaves represent 4KB pages in a 2MB address block. At each level, values of the child node are accumulated in the parent. The prefetch region is the largest such region that exceeds the threshold. All nodes in the prefetch region will be set to their maximum value, and leaf pages will be physically fetched. The real implementation has nine levels total.

Figure 2.3 shows a sample prefetching scenario with only four levels, and ignoring big page upgrading. Within this figure, one more fault is sufficient for fetching the full region; requiring

approximately five faults to fetch the full region. However, as the number of levels scales, single faults can cause a cascade effect that fetches large amounts of data with far fewer faults - an additional fault could cause an entire fifth level to be fetched. This is further enhanced by the page upgrade stage, as each fault fetches the entire corresponding level five subtree. In this scenario, fetching the entire VABlock only requires five faults, if they all belong to different level five subtrees.

2.3.2 Oversubscription and Eviction

UVM implements an eviction mechanism for the case where device memory is fully exhausted. The eviction mechanism is triggered whenever the driver attempts to allocate memory for a VABlock that does not have memory reserved on the GPU already. Evictions are performed at the VABlock level, mirroring allocation. When evicted, any modified pages are copied back to the host, and the physical memory allocation for the VABlock is released.

The UVM driver uses least-recently-used eviction from the perspective of the UVM driver. The LRU list is updated when a fault is handled from a VABlock. When eviction is required, the VABlock at the end of the list is evicted and removed from the list. The LRU list is only updated when the host is made aware of a memory access, which is only possible when a page fault occurs. By default, the host has no information about the frequency of non-faulting accesses on the GPU. Because of this, a VABlock that has been fully migrated to the device can never move up in the LRU list, even if the VABlock is under consistent heavy use on-device.

2.4 Related Work

In this section, we discuss the prior work on UVM and their contributions. We divide these works into two main categories: (1) UVM performance studies characterize or analyze the high-level performance attributes of UVM, and (2) improvements, extensions, and alternatives to UVM, where improvements and extensions change or extend some core functionality of UVM, and alternatives are based on paradigm shifts away from modern UVM.

2.4.1 UVM Analysis and Characterization

The analysis portion of our work focuses on the low-level performance of UVM, including systems-software and interconnect technologies. Several works have examined UVM performance

in terms of its performance vs. traditional, programmer-managed memory in synthetic, micro, and application-level benchmarks. These studies form the initial motivation for our work, but are not deep enough to root out the causes and potential improvements to UVM.

Landaverde et al. do a performance study on the Rodinia [9] benchmarks using UVM [26]. They introduce a version of these benchmarks modified to utilize UVM for comparison against the original benchmarks. Benchmarks are compared by absolute runtime as well as the breakdown into kernel time and data transfer time. In general, this work provides the high-level conclusion that UVM frequently harms application performance. This provides a high level understanding of performance, but does not capture the influence of systems software and overhead, which we show to be significant.

Following work investigated UVM with "advanced features," namely oversubscription, programmer-defined asynchronous prefetching, memory access hints, and ATS. Chien et al. analyzes memory access hints, programmer-defined prefetching[2], and oversubscription [11]. Gera et al. [18], Wang et al. [41], Sabet et al. [37], Min et al. [28] study the use of memory access hints and problem-specific kernel optimizations to improve the performance of graph traversal on GPUs. Finally, Xu et al. even utilize machine learning to try to predict appropriate cases to apply memory hints [42].

Similarly, Knap and Czarnul study programmer-defined prefetching and oversubscription [25], and Gayatri et al. study programmer-defined prefetching with and without ATS enabled [17]. Lastly, Gu et al. introduce UVMBench, which are a combination of the modified Rodinia benchmarks and several targeted machine learning micro-benchmarks for the express purpose of studying UVM performance vs direct performance with and without programmer-defined prefetching [19]. We summarize their findings here:

Hints and Prefetching. These studies find that performance can be improved in situations where these optimizations can be suitably applied. For programmers, these optimizations generally require a very deep understanding of the problem, and become increasingly difficult for the programmer to apply as the complexity of memory layout increases - situations where UVM is meant to simplify memory management. A machine learning approach can potentially be utilized to do apply hints on behalf of the programmer, but requires extensive performance profiling. In general, the study offers high-level insights to these optimizations and provides some speculative insight

[2]Note that this programmer-defined prefetching is an explicit CUDA runtime call for asynchronous data transfer, and not the implicit runtime prefetching performed transparently by the UVM driver.

into the internal workings of UVM, and show that extensive programmer optimization can improve UVM performance at the cost of programmer knowledge and effort. However, these optimizations are limited to read-only duplication, remote mapping, and explicit prefetching; they do not address the root performance issues inside of UVM.

Oversubscription. These works find that oversubscription can significantly harm application runtime but can be somewhat mitigated with hints and prefetching. This serves as motivation to understand and improve oversubscription as a key feature of UVM, to reduce programmer effort and improve situations where these optimizations are not effective or available.

ATS. The hardware acceleration for page translation provided by ATS gives greater performance, and our first insights into how HMM will work. This high-level understanding is tangential to our work, as ATS moves some components to hardware but does not fundamentally alter the inner workings of UVM.

Notably, some of these works and the case-study by Nadal-Serrano and Lopez-Vallejo [30] show improvements to performance when switching from direct management to UVM. These situations are not deeply studied in-context, but can happen in cases where kernels do not utilize all allocated GPU memory; in these cases, UVM implicitly only migrates the necessary data, whereas the programmer-managed memory applications will generally migrate all data.

In general, the focus of our work is to provide lower-level insights to the fundamental costs of UVM than these prior works, as well as focus on a systems-software-based approach to improving performance and reducing overhead to improve performance without requiring, but possibly in addition to, heavy memory-management effort on behalf of the programmer.

2.4.2 UVM Improvements, Extensions, and Alternatives

Several works have focused on improving UVM or extending UVM with new features or modular replacements for existing features. Extensions and improvements to UVM have been proposed, but most performance-targeted work focuses on changing the runtime prefetching/eviction mechanisms by migrating them into hardware and altering the algorithms, although some simply extend the functionality of UVM without altering the base. Alternatives to UVM generally require full paradigm changes to move away from the demand-paging virtual memory system towards a hardware-based shared memory system; we do not focus on these extensively because our work is centered on systems in-use today.

2.4.2.1 Extensions and Improvements to UVM

Yu et al. propose coordinated page prefetch and eviction, a method for coordinating the prefetching and eviction method within UVM [43]. They simulate hardware extensions that allow for tracking access patterns using a pattern buffer, enabling pattern-based prefetching and dynamic eviction algorithm switching based on access pattern. This is tangential to the existing UVM implementation, requiring significant migration of features from the UVM driver into the hardware.

Ganguly et al. propose a runtime extension utilizing GPU page access counters to take an adaptive approach to page migration when memory is oversubscribed [16], and similarly Chang et al. proposed a compiler-instrumentation approach to page counters to improve UVM eviction [8]. These approaches can pin data on the host for remote RDMA instead of migrating when the access pattern is poor for migration. This is a highly related subset of the UVM system, but their methodology is based on hardware simulations of the involved components. Ganguly et al. also propose moving the prefetching and eviction mechanisms into hardware, and altering their algorithms to provide more cohesive strategies between prefetching and eviction [15]. In our work, we confirm the assertion that UVM prefetching and oversubscription can conflict on an existing system with lower-level analysis of the performance implications. For analysis and performance improvements, we focus on the full scope of UVM which is inclusive of prefetching and eviction, and focus on methods available to improving the underlying costs of UVM runtime/driver software and its performance on modern systems.

Kim et al. propose batch-aware unified memory management, a hardware-software solution that (1) extends the capacity of existing hardware to allow greater latency hiding for page-level migrations, and (2) making eviction asynchronous so that it does not limit paging activity [24]. They use specific batch-level latencies to justify the need for these extensions. Our work shows a far more extensive focus on the costs of page fault batches inside of UVM, as well as their variation between applications and the root causes of these latencies. While our work agrees with the finding that asynchronous eviction would be helpful, it is not the focus of our work.

DRAGON extends UVM to use hardware storage as an additional layer of storage backing host memory [27]. The main benefit to this is an extension of the amount of physical memory available, furthering the capabilities of oversubscription. This is a feature extension that studies its own performance, and would stand to benefit from improved UVM performance and analysis.

2.4.2.2 Alternatives Architectures

Griffin is a series of hardware modifications that enable smarter locality detection and page migration [5]. The end result is similar to effective application of UVM memory access hints, but transparent and systems-oriented. This is a full UVM system-overhaul, and requires full vendor adoption.

Mosaic offers hardware and runtime extensions to enable dynamically-created huge pages for more effective TLB utilization on GPUs [4]. The authors propose this work as concurrent to UVM and other demand paging solutions, although UVM does implement an adjacent approach using VABlocks.

MGPU-TSM argues that RDMA costs and demand paging on modern systems are too expensive, and that host architecture should be redesigned such that GPUs have the same first-class direct access to system memory as the host CPU [29]. This work would require full vendor adoption from both GPU and host component manufacturers, and does not seem to be the rising trend.

Akshintala et al. introduce "Talk To My Neighbors" transport, a scheme applying task-based programming to memory management [2]. This scheme uses special hardware called "transporters" to appropriately manage the location of memory based on a programmer-specific task execution "plan." Similar to others, this approach requires a significant overhaul to system design with a specific programming and memory management model in mind.

2.4.3 Summary of Novelty

In our work, we take a holistic approach to analyzing and improving UVM, focusing on the systemic causes of performance issues in modern UVM. Prior analysis has not identified the root causes of overhead at the systemic level, which must be understood to build a foundation for improving these issues. Without this foundation, prior work on improvement has focused on ad hoc components of UVM such as prefetching and eviction, but do not address the root performance issues within UVM. Alternative system designs are able to forego this issue entirely, but require vendor adoption of hardware alterations and are therefore not applicable on existing and near-future HPC systems. Our holistic analysis enables us to create specific, targeted improvements to UVM that are applicable (1) on today's systems, (2) alongside other future architectural modifications, and (3) with upcoming software systems such as HMM.

Chapter 3

UVM Workload Generation and High-Level System Performance

In this chapter, we seek to understand how the UVM workload is generated and gain a high-level understanding of the system performance. These two aspects give us a high-level understanding of the UVM system and guide our deeper, batch-level analysis in later chapters.

3.1 Experimental Environment

All experiments in this work are performed on a Titan V GPU with 12GB HBM2 memory using CUDA 11.2 and NVIDIA Driver version 460.27.04 on Fedora 33, kernel 5.9.16200.fc33.x86_64. The system has an AMD Epyc 7551P 32-Core CPU with 128GB of memory.

Data is collected through a modified version of the UVM driver, which is distributed as source code alongside the NVIDIA driver. We modify the UVM driver into two versions. One is able to log per-fault metadata which we use to gather 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, where we collect the data using a custom logging tool that is more reliable than `dmesg` for large quantities of data.

Benchmarks used for evaluation include two synthetic benchmarks including a regular and random access pattern, cuBLAS SGEMM [31], Stream (triad-only) [14], Tealeaf [13], HPGMG [38], forward and inverse cuFFT [32], and a cuSparse kernel that converts a dense matrix to a sparse matrix and preforms a sparse matrix multiplication [10]. All applications were developed by the authors unless attributed, in which case the authors ported these applications to use UVM with the exception of HPGMG.

3.2 UVM Workload Creation

In this section, we focus on revealing how the UVM workload is generated. This workload is tied to NVIDIA GPUs, although there should be similarities across similar classes of hardware. We identify the following characteristics in faults generated by UVM.

- The maximum number of outstanding faults in the fault buffer is limited on a per-µTLB, and sometimes per-compute-unit basis.
- Faults occur quickly, leaving no overlap between GPU and CPU activities.
- Data dependencies within generated code may limit fault generation even when required addresses are known.

Using this information, we can draw several conclusions about hardware utilization and limitations, and the features of the driver workloads. We also gain insight into the fundamentals of how fault batches are generated.

3.2.1 Formation of GPU Fault Batches

In UVM, the **fault batch** is the fundamental unit of work and responsible for overall UVM performance. Prior work has shown that the time spent in servicing batches contributes a significant portion of the runtime for UVM-based applications and causes slowdown [24, 3]. We begin by examining how batches are formed using targeted examples to gain in-depth understanding.

To understand how faults propagate to the GPU fault buffer and eventually form a fault batch, we examine a simple vector addition kernel using UVM for memory management. As shown in listing 3.1, each thread performs the computation $c = a + b$ for a unique index. Unique to this

kernel is that each thread separates its access by a full page, to give us a more comprehensive view of faulting behavior. This operation is performed three times for three different pages by each thread to verify the consistency of fault behavior and demonstrate some faulting properties.

Listing 3.1: Vector addition kernel using first float of each page.

```
1  #define FPSIZE 512 // 4096 bytes / sizeof(float)
2  #define TSIZE 32 // total # threads
3  __global__ void foo(float* a, float* b, float* c) {
4    uint tid = blockDim.x * blockIdx.x + threadIdx.x;
5    size_t page0 = tid * FPSIZE;
6    size_t page1 = page0 + (FPSIZE * TSIZE);
7    size_t page2 = page1 + (FPSIZE * TSIZE);
8    c[page0] = a[page0] + b[page0];
9    c[page1] = a[page1] + b[page1];
10   c[page2] = a[page2] + b[page2]; }
```

Listing 3.2: Annotated SASS assembly corresponding to line 8 in listing 3.1.

```
—— snip ——
LDG.E.SYS R9, [R2] ; <— a[page0]
LDG.E.SYS R0, [R4] ; <— b[page0]
FADD R9, R0, R9 ; <— scoreboard stalls for R9, R0
STG.E.SYS [R6], R9 ; <— c[page0]
—— snip ——
```

We start by examining the basic characteristics of batches and executing this simple vector addition code with a single 32-thread warp. Figure 3.1 shows the faults in the order they occur and separated by batches. For each of the three additions, faults corresponding to the vector-addition access pattern perform two reads per thread from vectors A and B followed by a write to vector C. The first batch contains exactly 56 faults, including all of the vector A reads and most of the vector B reads.

We draw two insights from this first batch of reads. (1) Each *thread* is capable of performing one or more memory read instructions resulting in faults without blocking, the same behavior of non-faulting CUDA memory accesses. (2) The maximum number of outstanding faults per µTLB is 56 on this architecture, which we have confirmed by comparing against larger and more complex

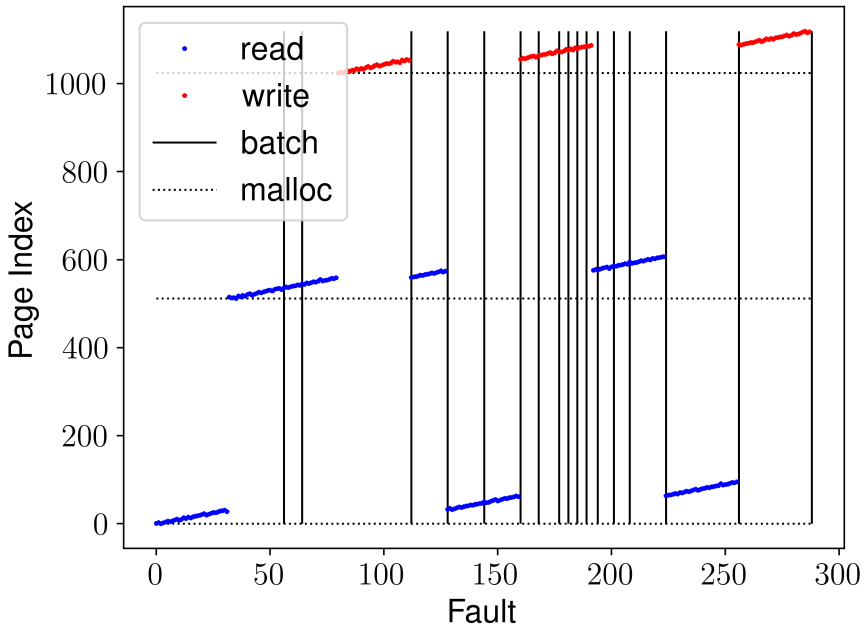


Figure 3.1: Relative time series of vector addition faults pulled from fault buffer, divided on the x-axis by *driver batch* and y-axis by *memory allocation*.

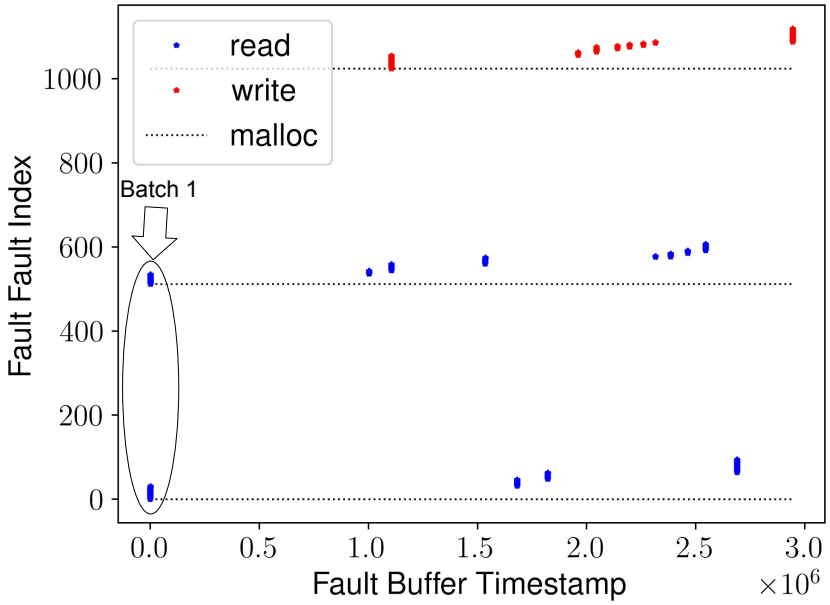


Figure 3.2: Fault data from figure 3.1 represented with real-time timestamp of when the fault arrived in the fault buffer. Faults clustered tightly vertically always indicate a batch.

examples. Figure 3.2 further shows that faults from the same warp happen in rapid succession when not held by hardware constraints, and that the full batch servicing time is short.

From the second and third batch of figures 3.1 and 3.2 we observe a subtle faulting behavior: no write accesses can execute until all 64 prerequisite reads have been fulfilled, even though the required memory addresses are known upfront. This can be traced to a subtle but consistent coding practice demonstrated in the resultant SASS assembly code in Listing 3.2 for one iteration of the vector addition. It becomes clear that the intermediate result of $A + B$ is required before the result is stored in vector C and the corresponding page fault is generated. For a coalescing version of the vector addition code, this implies that each faulting warp (or block) requires at least two full fault batches to complete its work, despite having the data requirements available upfront.

From this example we can also infer that in addition to the µTLB fault limit, there is an additional fault rate throttling mechanism preventing a single SM from creating too many faults. In figure 3.1, several batches consist of a small number ($<< 56$) of faults, even though there is no data dependency blocking the issuance of faults. These small number of faults are due to the presence of a rate-limiting mechanism on SMs. This inference is consistent with the original proposal of a far-faulting mechanism [45].

We demonstrate that (1) these limitations are tied to the µTLB level, and (2) faults are inserted quickly and are not in a data-race with the UVM driver, using instruction-level prefetching. Instruction-level prefetching can escape both limits on the number of faults and rate throttling. The compiled PTX high-level assembly code includes a set of prefetching instructions, such as `prefetch.global.L2`, which prescriptively prefetches data from global memory to the L2 data cache. As with typical memory accesses, if the data is not present in global memory then a page fault is triggered. Prefetching is unique in that it does not require the register scoreboard, thus presumably avoid triggering the aforementioned limitations. Figure 3.3 shows the resulting batches, where vectors A, B, and C are prefetched upfront. A single warp is able to generate up to 256 faults in a single batch, capped by the software batch size limit [1]. This far exceeds the prior per-SM fault generation capabilities, confirming our prior assertions about code limitations fault-throttling.

In table 3.1, we examine how these fault-limiting components scale to more realistic workloads. Using data collected from the GPU page fault buffer, we identify the SM originating each fault within a batch. Generally, each batch contains faults from nearly all SMs on the GPU. Depending

[1] Faults occurring beyond the batch size limit are dropped by the driver, and therefore not shown.

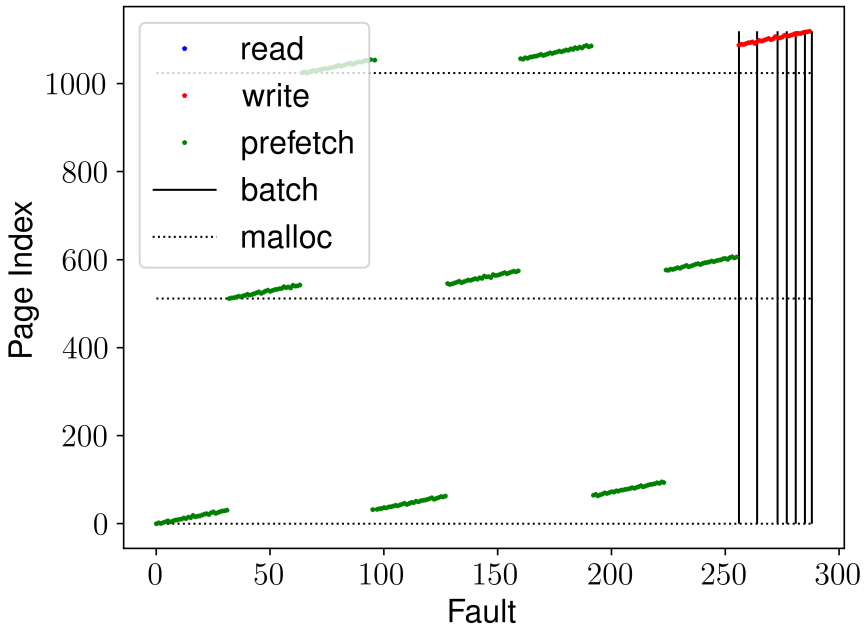


Figure 3.3: A single warp is capable of generating faults up to the batch size limit using prefetching. Prefetching is not beholden to outstanding fault limitations, and does not trigger fault-throttling mechanism.

Table 3.1: Per-SM Batch Source Statistics

Benchmark	Avg Faults/SM	Std. Dev.	Min.	Max.
Regular	3.06	0.43	0.09	3.20
Random	3.03	0.52	0.01	3.20
sgemm	0.85	0.60	0.01	3.20
stream	0.75	0.09	0.05	1.36
cufft	0.91	0.13	0.01	1.88
hpgmg	0.41	0.10	0.01	2.65

Each metric is “ averaged” among the batches. Faults are consistently sourced from a wide spread of SMs and faults-per-SM are handled in each batch.

on the application, there may be more than one fault, but no more than a few; each batch represents a combination of work across the GPU SMs. This behavior is consistent with the fault generation and rate limiting behavior discussed previously, and it shows that SMs are served relatively "fairly."

Overall, this experiment indicates that, at scale, batches contain faults from many SMs due to a combination of rate throttling issues and code generation based on operations that take places between data accesses. In the next chapter, we look at how characteristics of the generated workloads influence overall performance.

3.3 UVM Baseline Performance

Now that we have established the method of workload generation for UVM, we shift to the baseline performance of UVM, i.e. UVM without advanced features such as prefetching and oversubscription. We start with an example of UVM-level first examine two simple kernels while varying the associated data size over different runs. The first kernel is referred to as a "regular access" kernel, in that each thread accesses exactly one page corresponding to the thread's global ID. This means that access is regular within a warp, block. The second kernel has each thread access a single, random, unique page from the global buffer. For these experiments, UVM prefetching is disabled. Figure 3.4 show the total kernel time as well as the breakdown of time spent inside the UVM driver. The total cost is relatively constant in the order of 400-600 µs for data volume less than 100KB, indicating there is base overhead associated with UVM. This is different from explicit memory management where the cost is initially negligible and grows with data volume. For larger data volumes, cost increases roughly linearly as data volume becomes larger. This corresponds to the roughly linear increase in the total number of pages and therefore far-faults. For random accesses, we find that the *replay policy* also begins to take a significant proportion of the runtime.

Pre/post processing is shown to be negligible in cost, but functionally important for the fault servicing and replay implementation. Pre-processing first gathers faults from the device, performs basic bookkeeping and logical checks, and sorts them into the appropriate VABlock bins. NVIDIA documentation indicates that the driver uses a circular device-side queue to store a fault *pointer* when a fault occurs [35]. The host can read these pointers, which subsequently point to locations in the global GPU fault buffer that contain the full fault information. The driver will generally read at least a full batch from the queue during every pass and cache the faults on the

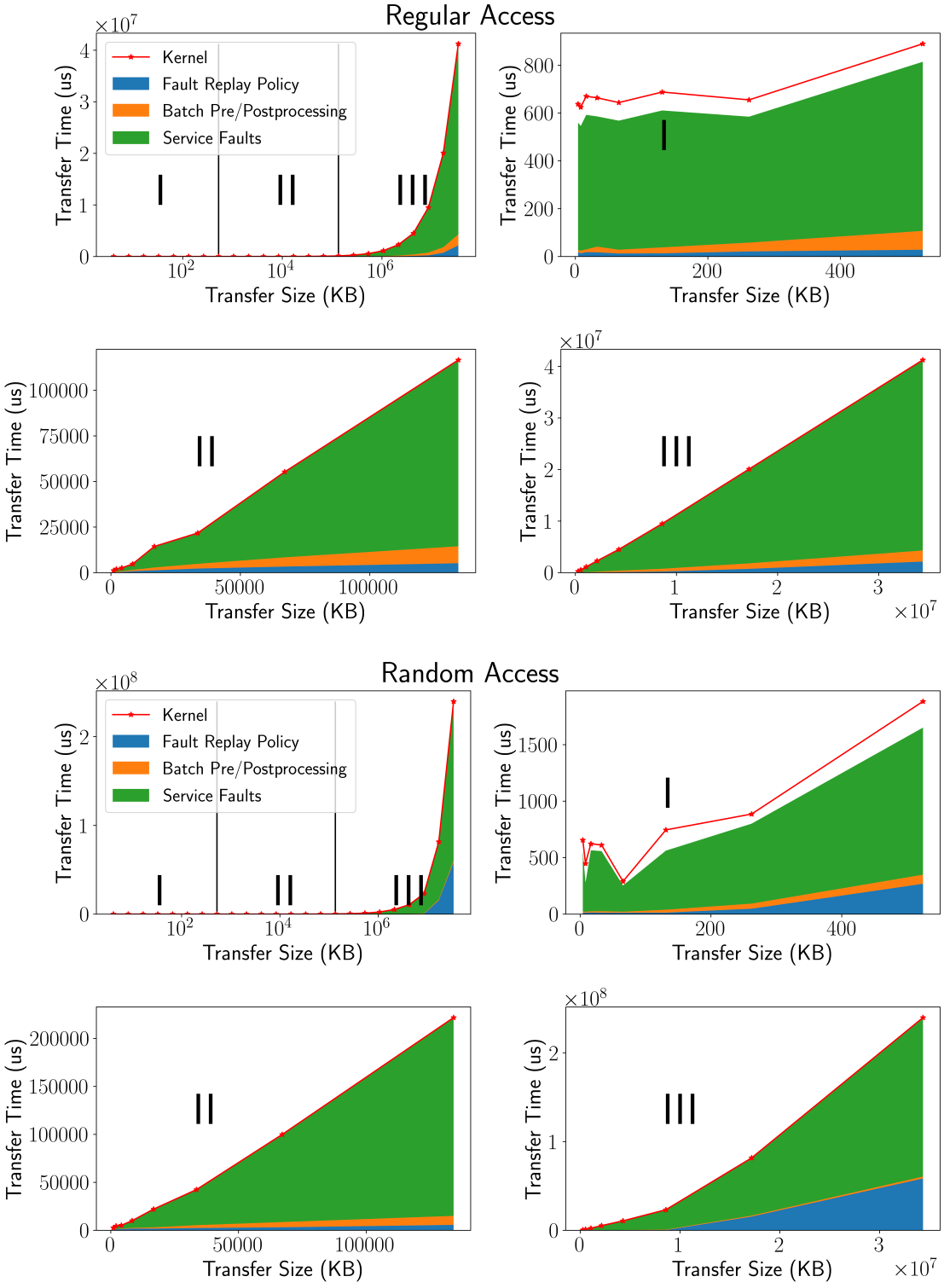
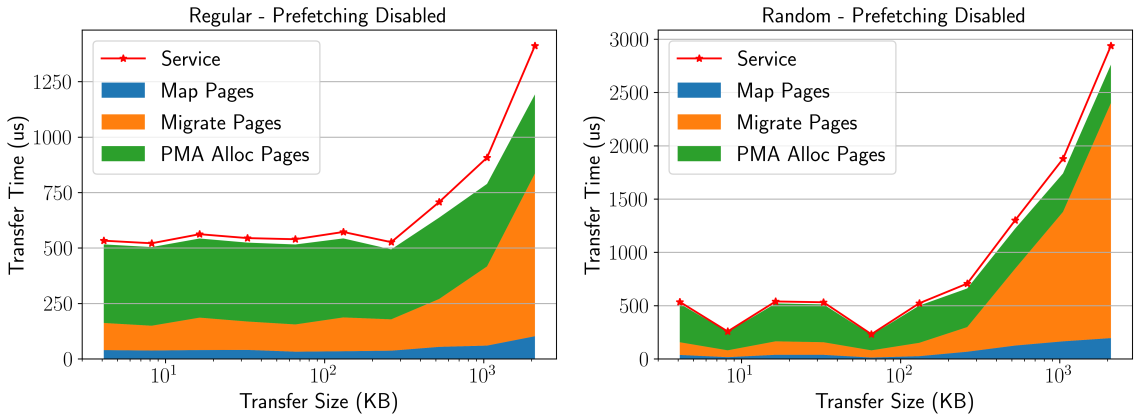


Figure 3.4: Fault cost scaling and breakdown at different magnitudes of scale for two access patterns on the same data size. Random tends to be slower, and has shifted proportions.



(a) (b)

Figure 3.5: Breakdown of the fault service cost. PMA Alloc Pages is a call into the proprietary NVIDIA driver to allocate physical memory. It is actually part of the migration process if required. Allocation seems subject to system latency, but allocations are usually over-provisioned to avoid multiple calls.

host to avoid having to make multiple remote updates to the queue. Faults may not be immediately available in the GPU fault buffer due to the asynchronicity. Thus the driver may need to poll the buffer until the appropriate "ready" field is marked true or may be able to begin processing on previously fetched faults. Sorting cost for batches is roughly constant due to the nature of sorting and the relatively small size of batches.

Fault Replay is a hardware cost, although it is dependant on *replay policy* and workload features, notably workload size. We can observe this trend at a very high-level in 3.4, increasing proportionally with the size of data for larger sizes. We examine these relationships more closely in the following section.

Fault servicing is a multi-step process that includes allocating physical space, zeroing out GPU pages, migrating data from the source to the destination, mapping pages and permissions, and a number of other tasks. We have created subcategories that cover the main costs. Figure 3.5 shows the cost distribution of service at small sizes showing our main categories: Map Pages, Migrate Pages, and PMA Alloc Pages.

Physical memory allocation accounts for a large but variable quantity of service cost. The UVM driver uses a physical memory allocator to track physical allocations on the GPU. Allocation is performed by calling into the main NVIDIA driver, which is not open-source. This makes it difficult to infer any hardware-level cost, but the cost seems sensitive to system latency. The allocator over-

allocates memory to cache it, knowing that the cost of each call is quite high. This over-allocation and caching causes the allocation cost to remain relatively constant and negligible at large sizes. This cost is actually contained within the greater "Migrate Pages" category, but is separated here as it is responsible for the "constant" dominating transfer cost within UVM at small sizes.

Page migration involves permission checking and updates, memory allocation and zeroing of newly-allocated memory, copying data from the source location to staging locations, and eventually issuing GPU instructions to copy data from the staging location to the final destination. Once data is staged on the destination device, page duplication would be broken and unmapped from source locations. The UVM driver initiates the memory copy command, and notifies the GPU to actually perform the data copy using DMA.

Mapping data includes updating the local and remote page tables and issuing appropriate memory barriers to ensure consistency on the GPU. While updating the GPU page tables is part of the cost here, the importance of this step is in bookkeeping and ensuring data consistency and integrity.

Naturally, the page migration cost is significant and dominating at larger data sizes. However, we require a deeper level of analysis, as the breakdown and distribution of constituent costs differs depending on a number of workload factors. With this high-level cost breakdown in mind, we further our analysis to the per-batch level with a specific interest in the operations within page migration.

Chapter 4

Batch-Level Performance Analysis

UVM fault batching is the UVM driver workload, and the overall performance is determined by how the driver chooses to handle this workload. For some applications, the driver workload is relatively small but must be handled before new work can be created. For applications that generate larger workloads, the driver is forced to make decisions about how to handle the workload appropriately. Interestingly, some applications fit both of these categories at different points in a single kernel, creating a complex and difficult-to-optimize scenario for the driver. We investigate several key costs dependant on workload features:

- Data movement: the amount of data migrated to the GPU can be a significant cost but is not the dominating factor.
- Fault duplicates: faults for the same address that appear in the same workload batch are partially mitigated within UVM but can otherwise have high overhead.
- Fault distribution/access pattern: the distribution of faults over 2MB VABlocks sets an underlying trend for performance variance.
- Host OS interaction: some components, such as CPU page unmapping, require the host OS and actually incur significant overhead on the fault path.

In this chapter, we explore how these characteristics influence batch performance and, in turn, overall application performance.

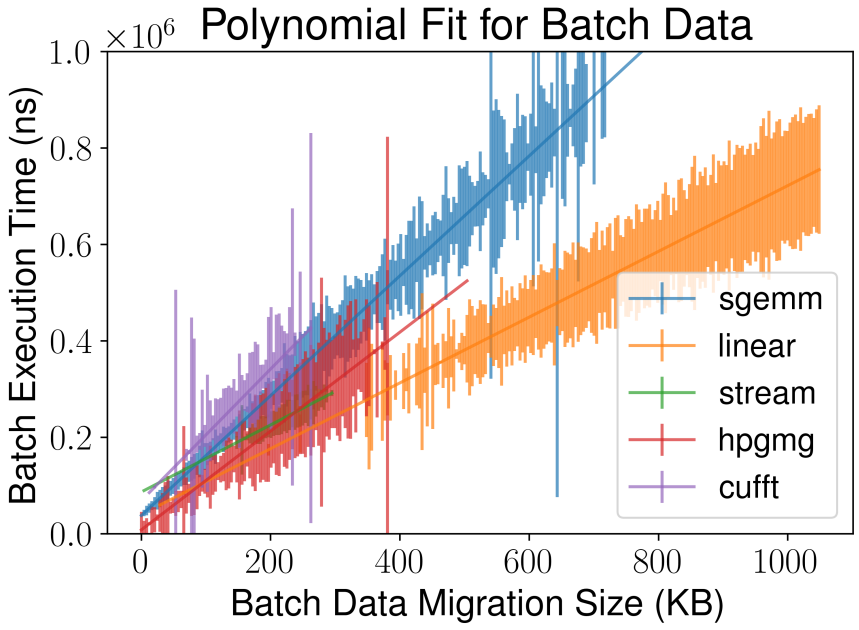


Figure 4.1: Best fit of batch sizes vs. data migrated for one run of several applications. Linear least-squares approximation with standard deviation per point.

4.1 Data Movement

Data Movement is the leading performance indicator in most UVM scenarios for a given batch. While other factors impact the overall performance, data movement is the primary purpose of the UVM driver and sets the trend for performance. Figure 4.1 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 high variance for each application.

Despite data movement being the key cost indicator, the actual data migration is not the primary cost in a fault batch. Instead, management is far more costly. We use the example of a moderately sized `sgemm` to demonstrate this point. Figure 4.2 shows that transfer time accounts for less than 25% of total batch time for almost all batches. This offers two insights: (1) Most batch time is spent in other components instead of the transfer over the interconnect and hardware command buffering systems. While faster hardware may benefit performance, the bigger issue is to ensure the driver efficiently utilize 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 cost and the sources of the

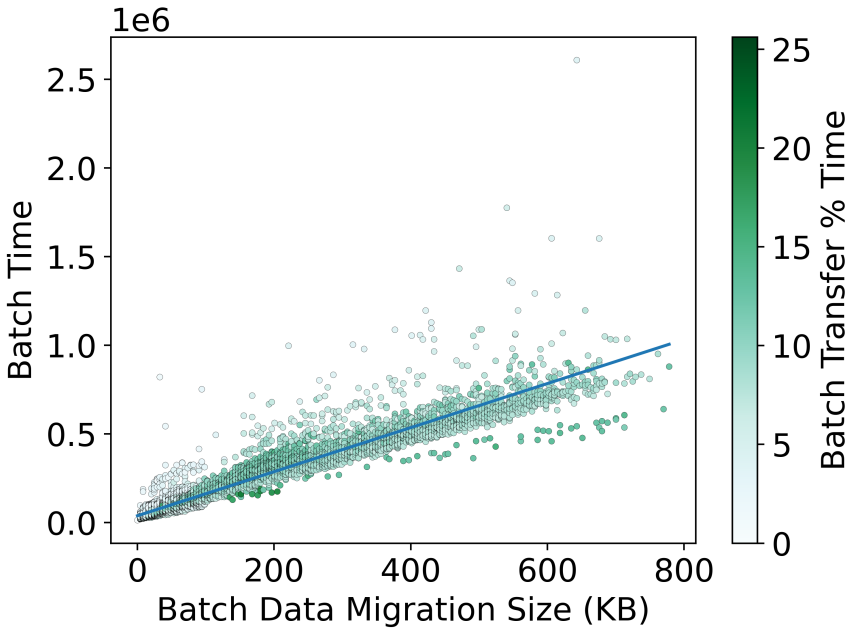


Figure 4.2: This figure shows the percentage of time spent per-batch performing data transfer relative to the total batch time for example `sgemm`. The batch time is approximately 25% at most, but typically far lower.

variance in the remainder of this section.

4.2 Duplicate Faults and Fault Replay

To further understand the characteristics of batches over an application's lifetime and examine the causes of variance, we look at the impact of *duplicate faults* on the overall batch performance. We demonstrate that (1) batches, the UVM workload, are not uniform across applications and non-trivial benchmarks have varying batch characteristics within a single run, and (2) we evaluate the performance tradeoff caused by duplicate faults when altering the batch size and flushing policy.

First, we demonstrate that the UVM workload is application-driven in terms of size and the number of duplicate faults. Figure 4.3 shows the actual batch size of all batches in an application execution as a time series, where (a) presents the raw batch size as pulled from the GPU fault buffer, and (b) the number of faults in each batch after discarding duplicate faults (faults that appear more than once). `sgemm` is typically far more complex than `stream` in implementation, and such complexity is manifested in the changes and "phases" of its batching behavior over time. For both

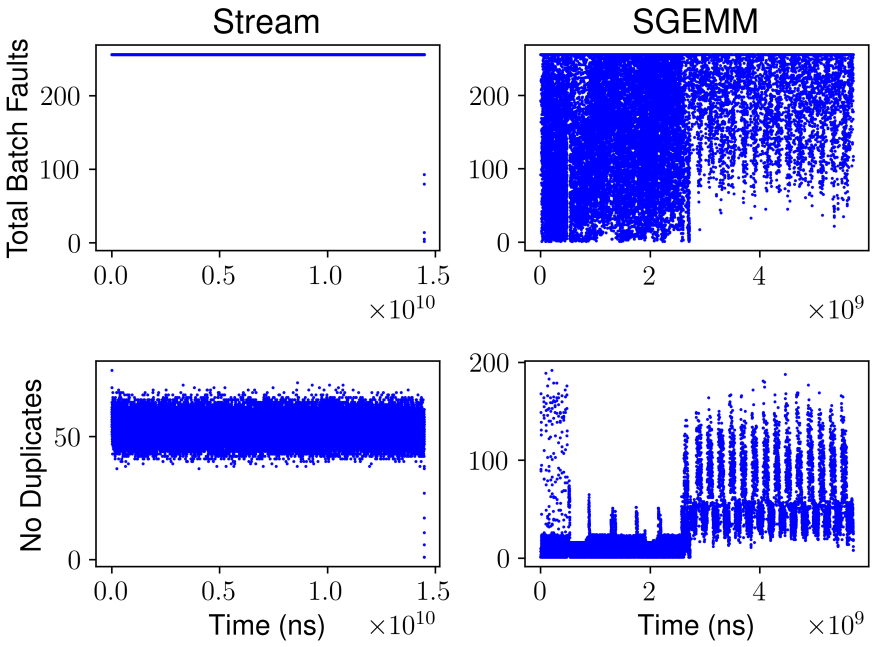


Figure 4.3: Time series of batches processed by the UVM driver pairing stream and `sgemm`. The first row is the total number of faults registered by the driver, and the second row is the total number of faults after duplicate faults (multiple faults to the same page) are removed. Duplicate removal is an early step the driver takes in fault processing.

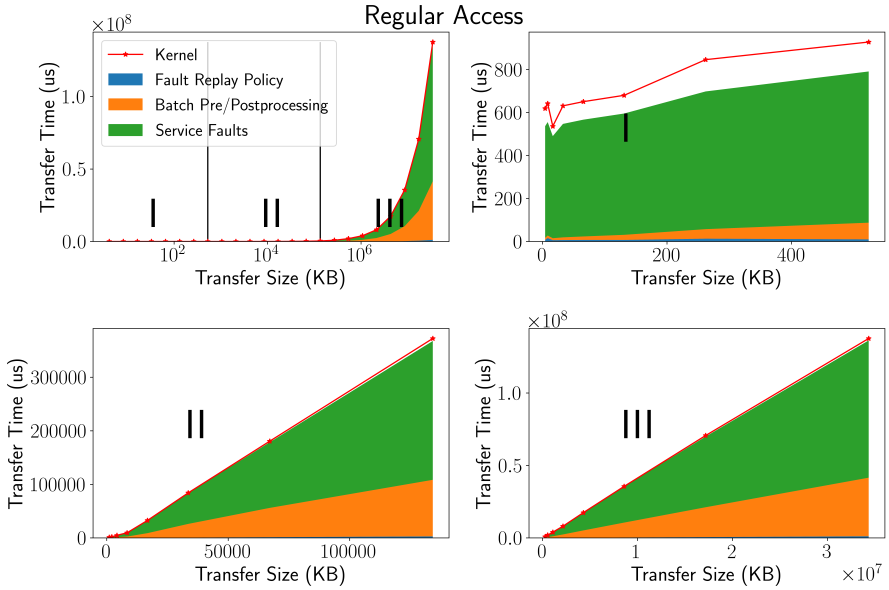


Figure 4.4: This figure demonstrates the same experiment as 3.4, except with the Batch Policy as the replay policy. Note that the replay policy cost is severely diminished, while the preprocessing cost is greatly increased.

applications, filtering out duplicates greatly alters the average size of batches, indicating a need to address duplicate faults. However, applications have varying degrees of complexity, and the impact of duplicates is not uniform even within the singular `sgemm` application. In the context of batch workloads, such non-uniformity explains portions of the variation in batch distribution previously seen in Figure 4.1, as duplicate faults do not contribute to the migration size but certainly account for a portion of overhead.

To give a high-level understanding of the impact of duplicate faults, we examine the impact of changing the batching policy from the "Batch Flush Policy" to the "Batch Policy." The primary difference between this policy and the default is that the fault buffer is no longer flushed after each batch. Any stale faults that were already serviced are left in the fault buffer to be serviced on the next batch. As expected, figure 4.4 shows the tradeoff in performance, causing a significant increase in preprocessing cost and overall cost over prior examples. The increased preprocessing cost indicates that flushing the buffer has the intended latency-reducing effect by eliminating duplicates without processing.

A finer-grain parameter impacting the number of duplicate faults processed is the maximum batch size. We investigate the tradeoff here by comparing the performance of various maximum batch

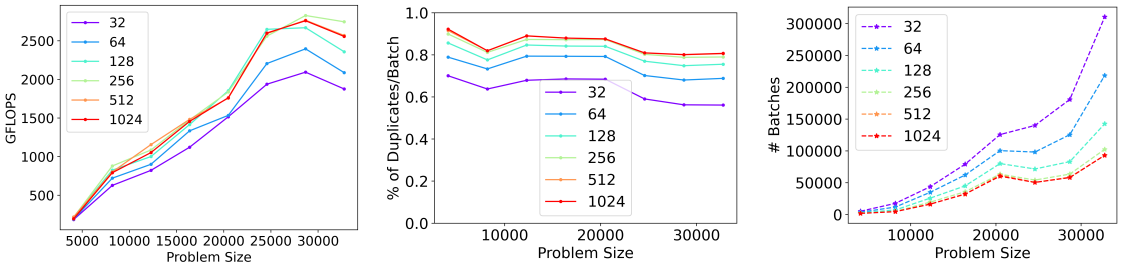


Figure 4.5: Batch size evaluation example with `sgemm`. While duplicate faults have some overhead, there is a strong correlation with the total number of batches required for processing. Batch sizes up to 6144 (max) are tested but are not shown as performance does not change.

sizes. Figure 4.5 shows the results where 256 is the default batch size. Critically, performance is generally greater with larger batch sizes, even though larger batch sizes have higher rates of duplicate faults. As larger batch sizes have smaller numbers of batches for the same problem size, we derive that the overhead of performing a batch is more costly than processing additional duplicates within each batch. However, increasing the batch size has diminishing returns. The maximum average number of unique faults-per-batch across all tested applications is on the order of 500 in our test regardless of the batch size, and increasing batch size beyond 1024 does not meaningfully affect the outcome because of this. The limit in the number of total faults available per-batch is a combination of (1) flushing the buffer between batches, and (2) the limitations on total fault generation described in the previous section.

4.3 Fault Distribution/Access Pattern

We examine fault distribution as the distribution of faults in a batch across memory, e.g. spatial locality at the page granularity. This plays a role in the observed performance variance, as processing is separated along memory boundaries.

Within the UVM driver, all operations are logically separated on VABlock (page-aligned 2MB) regions, making VABlocks a large source of performance variation. Figure 4.6 shows batches colored by the number of VABlocks that require data transfer within the batch. While each batch is subject to other sources of variance, one major trend is that, for batches with similar workloads, more VABlocks incur greater costs and cause greater performance variation. This is consistent with our earlier observation that VABlocks within a batch are handled independently.

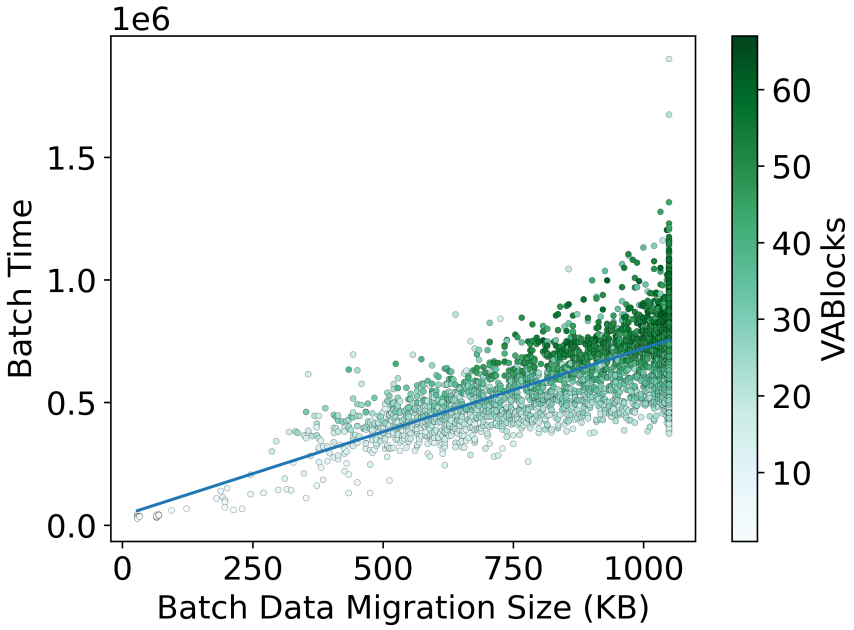
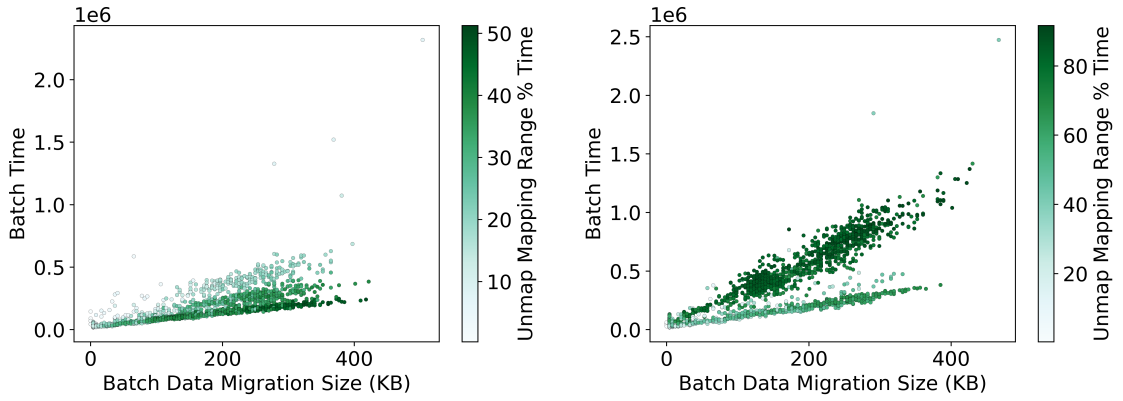


Figure 4.6: Batches sorted by to-GPU data migration size. Batches are logically divided into VABlocks, so each VABlock adds another layer of overhead. For a given migration size, higher cost is associated with more VABlocks.

Table 4.1: Per-VABlock Batch Source Statistics

	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
hpgmg	2.39	13.62	15.72	1	212

Average number of VABlocks in a batch and summary statistics of faults per VABlock. These statistics give us a general idea about the access density to specific VABlocks during each batch, and show that there is wide variation between each VABlock.



(a) (b)

Figure 4.7: These figures show two cases of the HPGMG application, where the percentage indicates the relative time per batch spent unmapping resident pages from the CPU as VABlock is logically resident on GPU. Figure 4.7a: single OpenMP thread; Figure 4.7b: one thread per CPU core. Unmapping CPU pages takes a larger percentage of time with CPU multithreading, indicating a link between host OS performance, CPU access usage, and GPU paging performance.

Processing each VABlock in parallel would be an intuitive optimization based on the driver design, but would be highly workload imbalanced due to the high standard deviation in per-batch VABlock representation. In table 4.1, 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. The root cause of this inconsistency is that, as discussed in the previous section, each fault batch contains pages from almost every SM on the GPU. Page faults in each batch originate from many different execution contexts, with only a few pages representing each SM. Random access is an exception to high variance, as it consistently has no locality within a single VABlock, but represents a very small workload per-VABlock.

4.4 Host OS Interaction

While we have thus far focused on the GPU faulting components of UVM, a critical component of the fault path involves migration of memory to the GPU from the CPU. The CPU component of UVM is built on top of the existing virtual memory system contained within the Linux kernel. Because of this, migrations are subject to latencies created 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 [38]. Figure 4.7 shows two examples of CPU-side behavior influencing GPU-fault performance outcomes. The two subfigures show the same problem with the same configuration, except (a) uses one 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.

Page unmapping represents a significant portion of execution time for many batches, as represented by the tone of color. Page unmapping is an operation in the existing virtual memory system on the host that is extended to support faults from GPU. Page unmapping is performed when a VABlock that is partially resident on the CPU is touched by the GPU through `unmap_mapping_range()` - all pages within the VABlock that are resident on the CPU are unmapped, as the CPU cannot use this data at the same time as the GPU per the UVM specification. The actual unmapping process relies on the Linux kernel virtual memory implementation. Interestingly, we observe that this cost is exaggerated in the case of OpenMP multithreading 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/or how threads are scheduled plays 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.This operation is performed by the host OS, and the costs likely stem from issues with virtual mappings across CPU cores, flushing dirty pages from caches, 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 CPU page unmapping on the fault path using host OS mechanisms, implying a similar cost could be applied to all devices using HMM [12, 23]. Design and implementation issues such as how unmapping takes place and if it needs to be performed on-demand deserve further investigation.

Chapter 5

Prefetching and Eviction in UVM

We now expand the scope of our analysis to include two UVM features enabled by default to support its use in real applications - prefetching and eviction. As prior work has shown, runtime prefetching is fundamental to allowing UVM applications to function at a level comparable to programmer-managed memory applications [3]. Oversubscription simplifies programming even further, allowing applications to work with out-of-core workloads, but typically at a high performance cost. In this chapter, we analyze workloads utilizing these two features using the fundamental performance mechanics discussed in the previous section, primarily identifying (1) algorithmic impact of access patterns on prefetching and eviction, and (2) how prefetching and eviction change both individual batching costs and the overall quantity and types of batches.

5.1 Prefetching

It is useful to look at *page-granularity access patterns* to understand how applications are perceived from the perspective of the prefetcher. Specifically, the driver does not see non-faulting data, and therefore the driver is oblivious to the full access pattern. Figure 5.1 shows the access pattern of several useful kernels. The x-axis represents the order of faults and y-axis represents the normalized location of the page in virtual memory. Points show an individual fault, while black lines separate the allocated memory regions (`cudaMallocManaged()`).

Each presented access pattern has unique elements to it that provide insights about how application-level access patterns are likely to appear to the driver, and we will discuss a few key

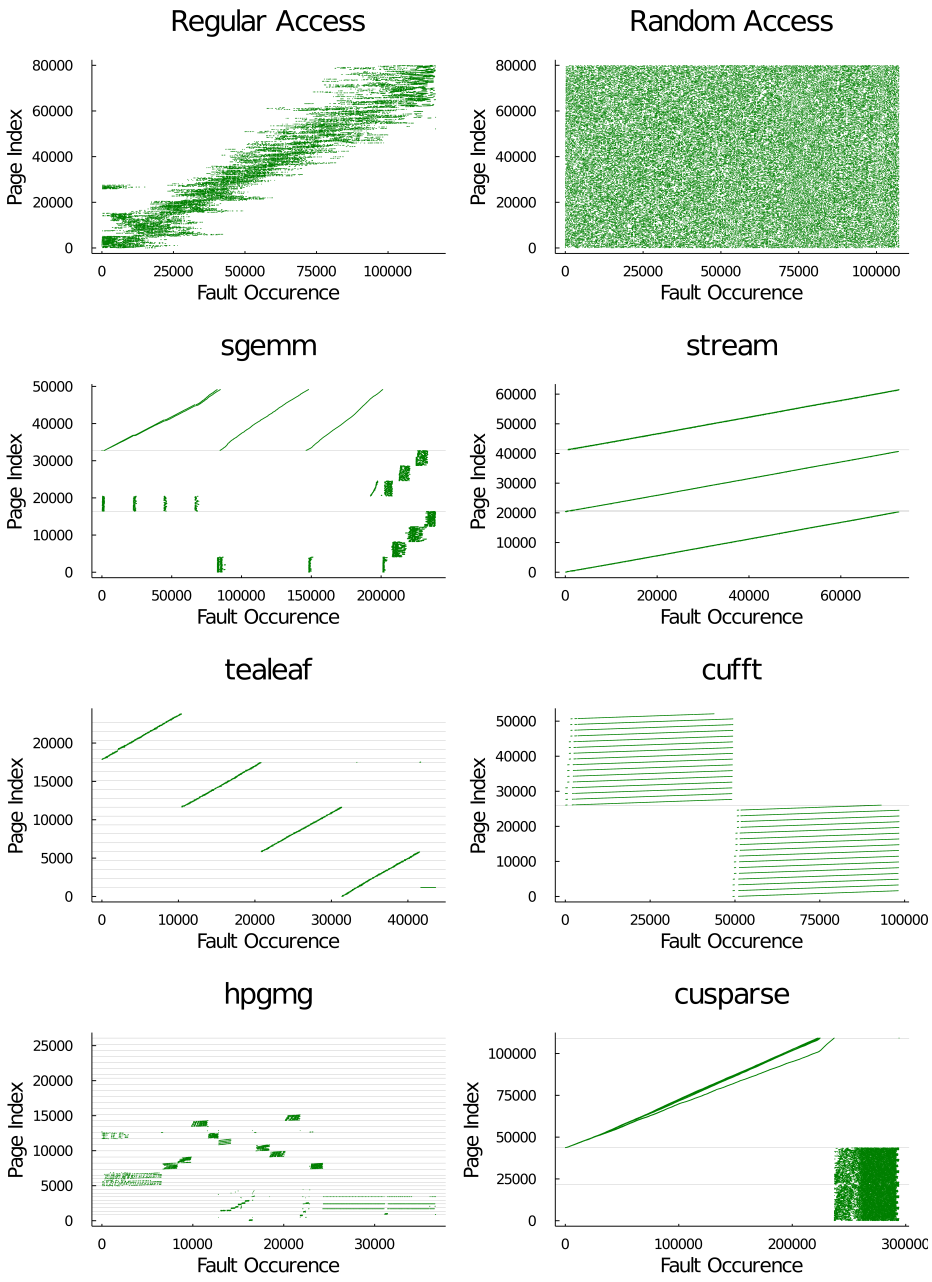


Figure 5.1: Application access patterns with prefetching disabled. The page index is the virtual memory page corresponding to the fault address, adjusted so that there are no gaps in the virtual memory space. Fault occurrence is the relative order that pages were processed by the driver.

Table 5.1: Application Fault Reduction

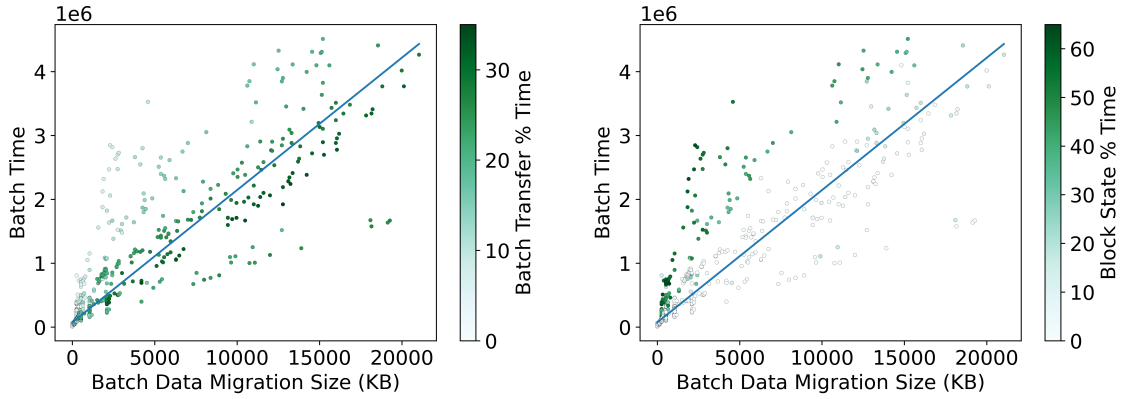
	total faults	faults w/ prefetching	fault reduction (%)
Regular	2493569	442011	82.27
Random	2522931	51558	97.95
sgemm	6522314	223998	96.56
stream	3721584	578884	84.44
cufft	101494	10074	90.07
tealeaf	1193619	394148	66.97
hpgmg	139785	50231	64.06
cusparse	2342572	611719	73.88

Fault collections with and without prefetching for relatively large undersubscribed problem sizes. Higher reduction is better, and is equivalent to fault coverage.

insights here. The "regular" access pattern described before demonstrates that the GPU scheduler will prefer lower-numbered blocks during access, but that there is no fixed ordering due to the non-determinism of the GPU parallelism. `sgemm` provides an access pattern similar to what we expect, but we must consider that the pattern does not show the heavy data reuse on taking place on the GPU. Stream offers an interesting contrast to the regular access pattern- the three-vector access pattern enforces a page-access dependency, enforcing a much more strict ordering of page fault handling than the regular access pattern. The `hpgmg` and `cusparse` benchmarks both show portions that mimic the random access pattern, characterizing the access behavior of sparse matrix representations.

Density prefetching ignores the precise ordering of page faults, which is critical for handling faults from many GPU cores simultaneously. This algorithm largely drops the timing aspect of prefetching. Instead, it utilizes the information already being tracked about page location to make confidence-based predictions about which data will be used. If a specific VABlock is saturated with faults over any length of time, the algorithm will confidently predict that the rest of that VABlock will also be used.

Fault Coverage. We analyze the impact of access patterns by examining our series of benchmarks. Referring to table 5.1, we find that the random access benchmark generates significantly less total page faults than the sequential benchmark, indicating the effectiveness of the prefetching in allowing the data to arrive sooner, and the effectiveness of scattering faults within a VABlock. The performance of the random access benchmark is several times worse for moderate data sizes, indicating that the additional faults and transfers from sub-optimal prefetching and driver overhead still harm the overall performance. In terms of the number of faults eliminated, **fault coverage**,



(a) (b)

Figure 5.2: An example of the previously-seen `sgemm` example after prefetching is enabled. The mid-range cost batches have been significantly reduced, reducing the overall time spent in the driver. There is a large range of values, with high-end outliers corresponding to negative performance impacts from creating and storing DMA mappings.

the prefetcher is still quite effective in both cases. For all benchmarks, at least 64% of faults were eliminated by enabling prefetching, indicating a high degree of effectiveness for fault coverage across a wide range of applications. Interestingly, in terms of fault coverage, `sgemm` and the random benchmark performed similarly in contrast to `hpgmg` and Tealeaf. All three of these applications contain random-like segments, but the prefetcher handles them very differently, indicating underlying timing and dependency constraints.

5.1.1 Batch Elimination and Performance

In figure 5.2, we see the results of prefetching on the previously-viewed applications. The number of batches is much smaller, reduced by 93% from the previous figure 4.2 of the same `sgemm` with prefetching disabled. However, some batches have highly exaggerated sizes due to large prefetching regions. The relative performance trend is similar to the non-prefetching trend.

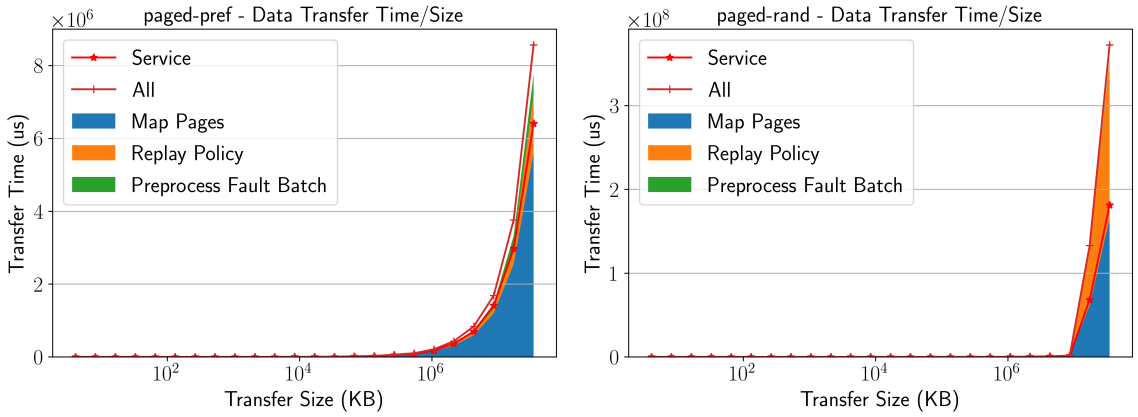
Many instances of very high-cost batches in this figure would have been considered outliers in the previous figures without prefetching. These batches are traceable to the behavior seen in figure 5.2b, showing that up to 64% of batch time is spent in GPU VABlock state initialization not present in other batches. This time is largely spent doing 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. The batches creating these mappings cannot be eliminated by prefetching, as they are compulsory when a VABlock is first accessed. However, not every batch requiring these DMA mappings has 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, but this low-level timing creates significant skew in the overall timing information and so it is not presented here.

The overall characteristic of prefetching shows that it makes a very similar tradeoff to batch size capping for duplicate faults; reducing the overall number of batches is highly effective in speeding up UVM, even when it means performing larger quantities of work in the short term. However, this serves to make the inconsistent DMA mapping cost a greater proportion of overall cost.

For workloads that do not exhibit eviction (undersubcribed), it would be preferable to use a more aggressive form of prefetching, as all data can fit in GPU memory. Since the data will most likely be used anyway, fetching it earlier better utilizes hardware resources by doing less but larger transfers, cutting down on overhead. The performance of using a 1% threshold rivals the performance of an explicit direct transfer of the full dataset, indicating that this should perhaps be the default setting for UVM when high performance is desired. This data is omitted due to space constraints.

In summary, for undersubscribed workloads the density prefetcher can range from effective to highly effective in terms of fault coverage, but can be difficult to code against. The prefetcher is somewhat fickle, requiring a small number of faults across the VABlock. If faults are not spread across the VABlock, they risk being considered duplicate when the pages are upgraded to large pages. The GPU scheduler and data dependencies in kernels play a role in the order that faults are issued, further complicating the issue of programming against the prefetching algorithm. However, general applications should expect good performance out of the prefetcher, and even better performance if aggressive prefetching is enabled.



(a) (b)

Figure 5.3: A breakdown of performance for oversubscribed problem sizes with prefetching enabled. In these figures, "Map" includes page migration and relevant costs. "All" refers to the kernel execution time.

5.2 Eviction

UVM includes a responsive eviction mechanism for the case where GPU memory is fully exhausted. The eviction mechanism is triggered whenever the driver attempts to allocate memory for a VABlock that does not have memory reserved on the GPU already, e.g. the first page fault. Evictions are performed at the VABlock level, mirroring allocation. When evicted, any modified pages are copied back to the host, and the physical memory allocation for the VABlock is released.

5.2.1 Eviction Cost and Patterns

Corresponding to access pattern, the eviction mechanism can evict data that is still being used. Because the LRU function is only aware of page-faults, it is possible that the "hottest" regions of data may also be the most likely to be evicted. The data would quickly be migrated to the GPU and then never again updated in the LRU list. We identify this behavior at the access-pattern level in figure 5.4, where data in the second memory allocation is evicted immediately prior to being paged back in, as the driver is ignorant to reuse on the GPU.

Relatedly, prefetching can fetch data that will not be used prior to eviction. This conflicting behavior has generally been noted by prior work [8, 16, 15]. As an example, UVM will move a minimum of 64KB per fault, either due to default host page size or due to the x86 page size

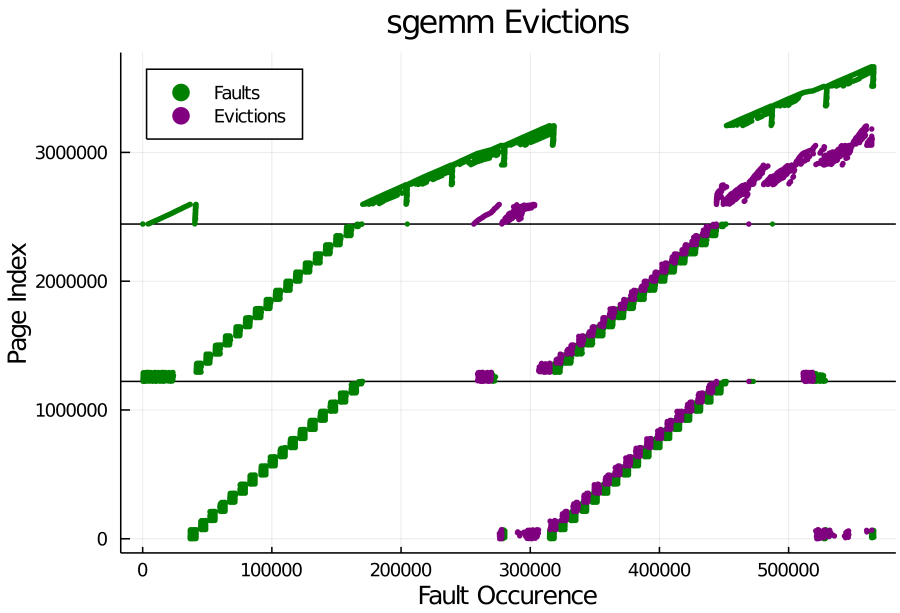


Figure 5.4: This figure demonstrates sgemm with a problem size that requires approximately 120% of GPU memory. Notably, we show evictions at the relative time step they are issued. Evict and re-fault is a worst-case performance scenario.

upgrade prefetching mechanism. With prefetching disabled, performance improves after prefetching is disabled, demonstrating the cost of prefetching to effective oversubscription. This can be seen directly by comparing figure 5.3 to figure 3.4 from the prior section.

Direct Costs. The direct cost of an eviction has two components. First, the eviction itself has the same components as a device-to-host fault for a VABlock not present on the host. The changed data needs to be migrated, involving data transfer, memory barrier, and page mapping/unmapping. Second, due to the locking scheme in the driver, eviction causes the VABlock faulting path to start over, as the faulting block lock must be dropped while the evicted block lock is held. The cost of a single eviction is not prohibitive compared to that of a page fault, but the number and size of evictions can increase the cost, as well as several indirect costs discussed in the next section.

Indirect Costs In terms of specific cost changes indirectly created by eviction, we observe the following: Proportionally, mapping pages for the random access pattern has only a slightly increased percentage cost with irregular pattern, but the overall cost of mapping pages is much larger due to the increased quantity of evict/map operations for small data sizes.

Corresponding to access pattern, the eviction mechanism can evict data that is still being used. Because the LRU function is only aware of page-faults, it is possible that the “hottest” regions

Table 5.2: SGEMM Fault Scaling

Size	# Faults	# Pages Evicted	# Evictions per Fault
29228	590719	0	0
30764	653504	0	0
32300	756502	79360	0.104
33836	1139293	2611200	2.291
35372	566018	3234748	5.714
36908	757216	6454152	8.523
38444	1827628	25170708	13.772
...			
47660	2697727	38092576	14.120

Evictions as problem size increases for SGEMM. Problem size is n for matrices A,B,C where size $= n^2$. Pages evicted are the number of pages that required explicit data migration between host and device. Correlating to figure 5.5, performance degrades as the number of pages evicted per fault increases.

of data may also be the most likely to be evicted. The data would quickly be migrated to the GPU and then never again updated in the LRU list. We notice this specifically in figure 5.4, where data in the second memory allocation is evicted immediately prior to being paged back in, as the driver is ignorant to reuse on the GPU.

Total Eviction Costs It is important to understand that the impact of eviction is at its greatest when data access is irregular. We can see this clearly in figure 5.3, where different access patterns show an order of magnitude difference in performance. This is due to the asymmetry between the size and granularity of eviction (VABlock) and the amount of data requested by a single fault (4KB). If the pattern is truly irregular, such as in our random benchmark, it is possible that only 4 bytes are required, but the full page fault processing must take place as well as the full 2MB data allocation and prefetching. This causes the GPU memory to become exhausted far quicker than required, as huge portions of memory are allocated but unused within a VABlock. The prefetching also causes larger-than-needed transfers for data that may not be used; a stark contrast to the huge fault reduction seen for random access patterns in the previous section. Profiling shows that for the 32GB problem size in our benchmarks, the regular synthetic benchmark requires only the base 32GB of data compared to 504GB for the random access despite having only 12GB of GPU memory emphasizes the importance of these memory constraints. The overhead of eviction mixed with the sheer number of additional faults and evictions from the poor access pattern account for the order-of-magnitude performance loss.

The overall cost depends on the measure of oversubscription as well as the access pattern itself. Since the GPU behaves as a cache, it follows similar locality principles as a traditional

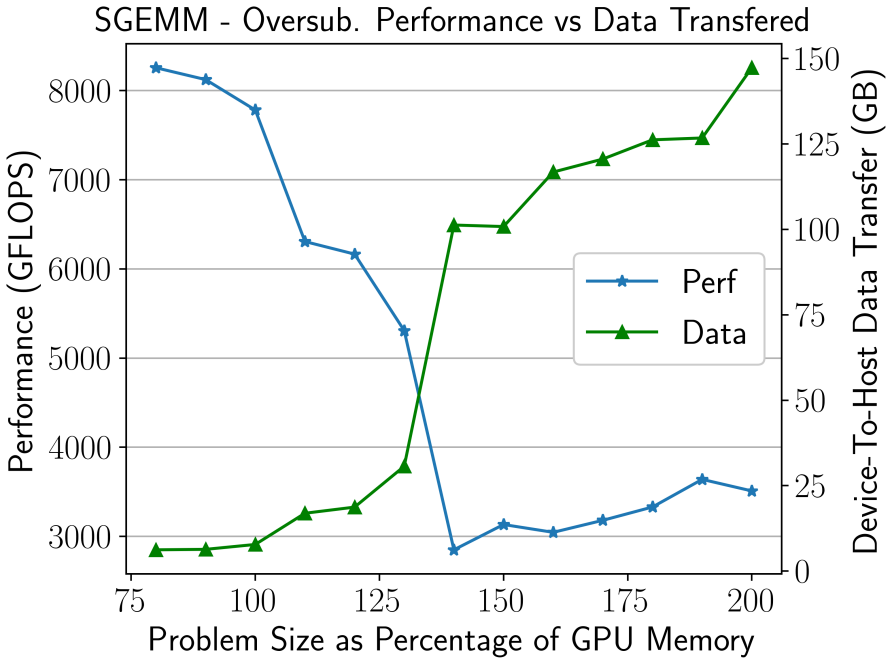


Figure 5.5: This figure shows the parallel increase in data requirement as compared to compute rate for the `sgemm` kernel.

hardware cache. This is demonstrated in figure 5.5, showing compute rate of SGEMM decreasing as oversubscription increases. The worst effect is noticeable when applications cross the threshold where local data no longer fits in-core, and data is evicted prior to being used. For example, performance degrades significantly after 120%, because the access pattern shows this evict-before-use behavior. Figure 5.4 shows the access pattern at this point, including evictions. Data evicted before use is a worst-case performance scenario. This cost correlation can also be seen in table 5.2, where we can see that the number of pages that require migration due to eviction rises as a partial indicator of performance.

5.2.2 Batching Impact.

In figure 5.7, we see an example where batches with the same number of evictions appear to show multiple "levels" of cost. The levels showcase an interesting component of the eviction mechanism. If a paged VABlock is resident on the CPU, requiring a call to the previously discussed `unmap_mapping_ranges()`, *and* the GPU memory is fully occupied, requiring an eviction, then both costs are accounted for in the overall time. In contrast, if a VABlock has already been made resident

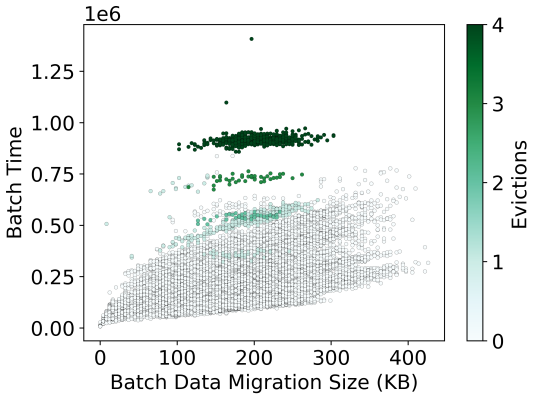


Figure 5.6: Batches by the number of evictions for an oversubscribed `sgemm`. All pages resident on the GPU are migrated back to the CPU.

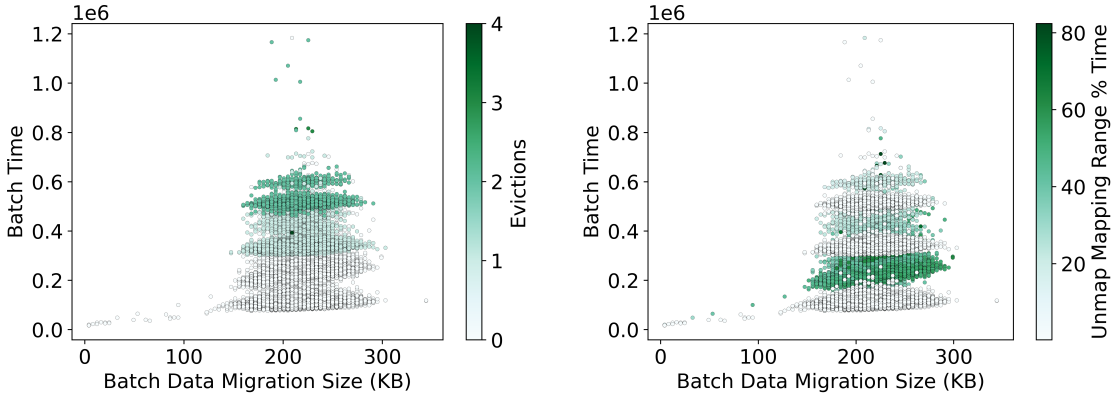


Figure 5.7: Oversubscription stream example. Batch characteristics determine the overall performance. LEFT: We can see multiple performance "levels" for the same number of evictions. RIGHT: we can see that a level may or may not include a portion of CPU unmapping.

on the GPU but later evicted, then it is not remapped to the CPU unless it is accessed by the CPU. If a VABlock was evicted once and paged back onto the GPU, then it does not have to pay the large `unmap_mapping_ranges()` cost for a second time, 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, where the lower "level" for the same number of evictions always has near-zero unmapping range cost.

5.3 Prefetching and Eviction

Finally, the combination of eviction and prefetching creates the most complex scenario. Prior work has shown that their combination can have a negative impact on performance for applications with irregular access patterns [24, 3, 44]. The relationship is somewhat indirect; since prefetching contained within a new VABlock cannot trigger eviction. However, data that is prefetched before use but must still be evicted later incurs an additional cost in both the initial migration and the later eviction. We evaluate this scenario by comparing prefetching enabled and disabled for the same application.

Figures 5.8 and 5.9 show an example of the `dgemm` with combined properties of eviction and prefetching in both the migration size-sorted plot and as a time series. The range of data transfers is still extended but not to the full 20MB as in the prefetching example alone; this can be attributed to reduced block access density for the larger problem size.

We examine each vertical pair of figures individually: (1) In Figure 5.8a, we can confirm that prefetching is still active and driving the larger batch sizes. Prefetching tends to happen earlier where VABlock are consistently resident on the GPU and subsequent accesses to the same VABlock can drive a strong prefetching response. (2) Figure 5.8b shows eviction ranges that are extremely similar to the non-prefetching data set, fitting into the same sizes and ranges. The eviction set has relatively low batch-sizes because they bring in *new* VABlocks, which have low access density when first paged in and trigger minimal prefetching. (3) In figure 5.9a, non-eviction batches including new VABlocks tend to have lower batch sizes, but have to 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. (4) Finally, in figure 5.9b, we observe that creating DMA mappings can still

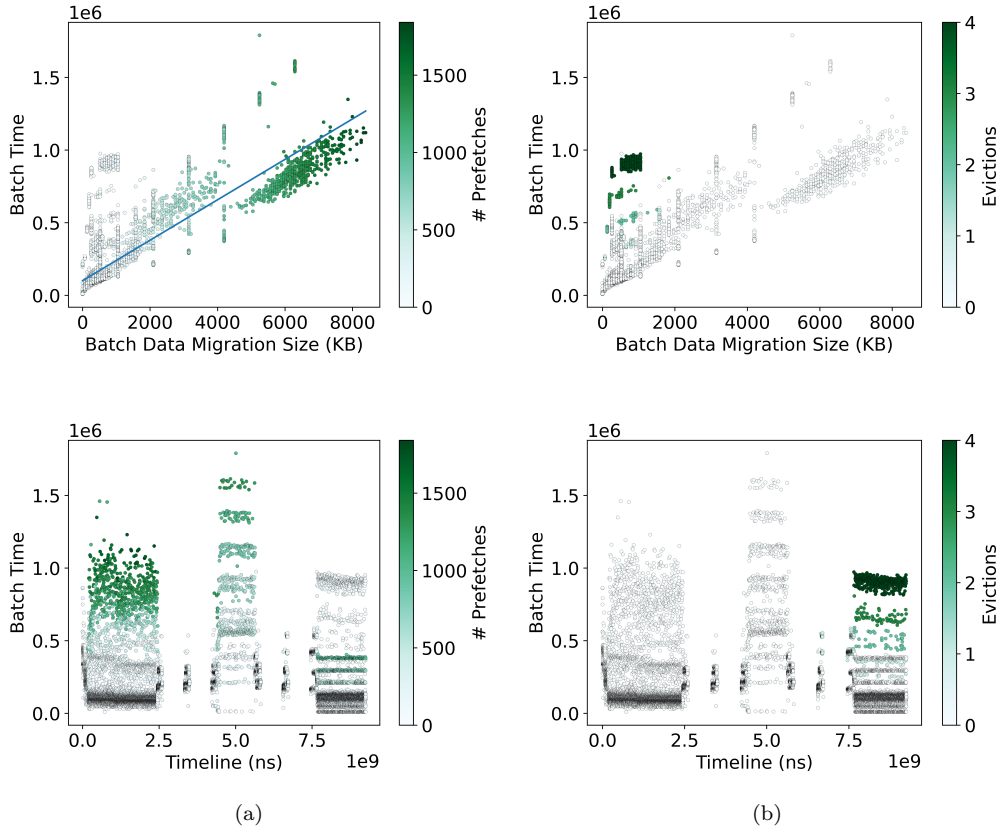


Figure 5.8: Batches from `sgemm` with high-impact performance events discussed in the previous section, sorted by data migration but also as a time series. The time series mappings for these events show when these events occur. Prefetching occurs throughout the execution but is limited when VABlocks begin to migrate off of the GPU. Evictions typically occur later in computation as it takes some time for the GPU memory to fill up.

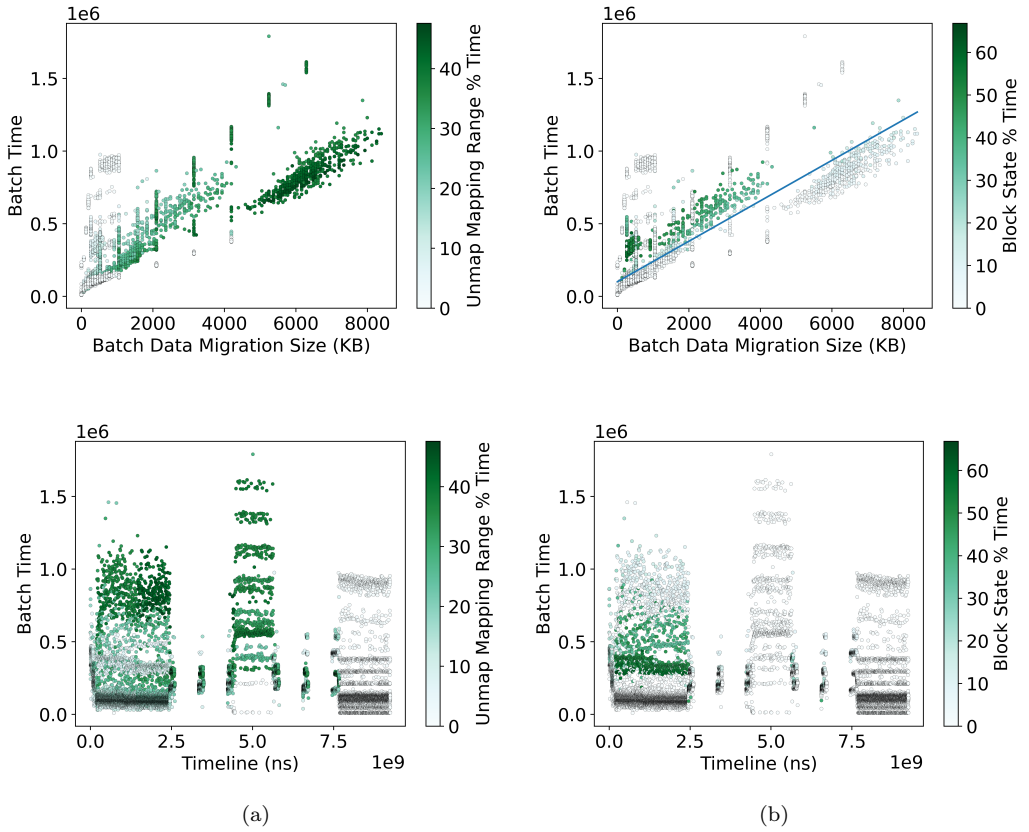


Figure 5.9: Additional breakdown of figure 5.8 Unmapping from the CPU occurs regularly throughout the application until every VABlock has received at least one page fault. GPU state setup (DMA time) also occurs throughout the full application, but does not always have excessively high overhead.

have high overhead, although it is intermittent. This figure suggests that the high overhead may be caused by growing of the underlying radix tree, but further investigation is required.

Overall, we confirm our intuition about *when* these batch features may occur, as well as confirm that many of the cost relationships we have discussed still account for a large quantity of runtime even with eviction and prefetching enabled. Additionally, we find that eviction costs are largely independent of the host OS performance problems discussed, and needs to be optimized independently. Prefetching is able to keep a large number of batches from suffering host OS issues, but does not mitigate the costs for the remaining batches.

Chapter 6

Optimization

In this chapter, we utilize our prior findings to make targeted optimizations to the UVM system. Enhancing UVM at the system level is significant because all implementing applications will immediately receive a performance improvement without code modification. The primary target of our optimization is host page unmapping for several key reasons:

- Unmapping is an important operation for host-device coherency, but is a realtime bottleneck.
- We have demonstrated that host-level parallelism designed by the programmer can inadvertently but significantly harm device performance because of page unmapping.
- Unmapping is functionality controlled by HMM in its implementation, meaning that poor performance will propagate to all future UVM-like implementations.

This makes page unmapping an obvious target for implementation, and has potential to improve implementations beyond UVM. In this chapter, we present *preemptive* and *asynchronous* methods for unmapping host pages and analyze their tradeoffs, benefits, and use-cases.

6.1 The Host Page Unmapping Problem

During a device-side page fault, UVM will unmap the VABlock containing the required page from the host if necessary prior to migrating the data to the device - a necessary but costly operation. This operation is important for two main reasons: (1) UVM is not designed to allow concurrent access on host and device simultaneously, and detection of this requires a page fault, and

(2) the system ensures that host TLBs no longer contain references to unmapped pages to prevent stale TLB access. Unmapping before migration ensures the consistency and coherency of data within UVM. However, we have previously demonstrated that unmapping these pages and flushing TLBs is expensive.

The impact of these operations affects (1) the consistency and turnaround time of individual system batch handling operations, and (2) can also be detrimental to application performance in certain common cases. (1) Pages are unmapped on-demand as they are accessed, causing an inherent spike in the cost of handling batches requiring unmapping operations. This means the first access is inherently more expensive. (2) The unmapping cost is 'reset' every time the data is required by the host, causing it to trigger again on the next device access. Applications that heavily utilize hybrid computing or require aggregation of data between multiple devices or nodes on host machines are subject to pay this cost frequently.

6.2 Preemptive Asynchronous Host Page Unmapping

We propose preemptively host page unmapping as a method to mitigate the cost of page unmapping on-demand. The key idea is to begin the unmapping operation prior to the data being required by the device, avoiding the need to unmap pages on-demand and batching unmapping operations. We additionally consider performing preemptive unmapping *asynchronously* on a background thread to allow other batch processing to take place alongside it. This allows us to hide the latency of these preemptive unmapping operations by performing them in the background while continuing other batch processing operations.

There are several considerations in applying preemptive and asynchronous unmapping: the appropriate granularity of preemptive unmapping, what conditions should trigger for this unmapping to occur, and preserving the correct semantics for avoiding stale TLB accesses.

Unmapping Granularity. Pages can be unmapped at any time from the host. However, if they are accessed on the host again before usage by another device, they must be remapped to the host. The behavior of an arbitrary application can not easily be determined prior to execution, so this case must be considered. In our method, we evaluate unmapping the entire unified address space (aggressive), individual memory allocations (moderate), and per-VA Block (default, conservative).

For many applications, we expect that unmapping the entire unified address space is appro-

priate. In contrast to host process address spaces, the unified address space tends to only include application data. Further, device kernels are likely to use *all* allocated memory in the unified address space, as generally only kernel-required data is allocated in the unified address space. However, this is dependant on the programmer and may not be appropriate for instances where not all unified data is used on the device. A more moderate approach unmaps on a per-allocation basis under the idea that if we have evidence that a data structure will be used by a device, it should be effective to preemptively unmap that data structure. This assumes that a memory allocation generally has some uniform purpose and that there is a degree of temporal locality within the memory location. This would prevent us from unmapping entire allocations that were not used for a particular kernel. We also include the existing VABlock granularity of unmapping for contrast as the lowest reasonable granularity.

Unmapping Conditions. Similar to unmapping size, we must unmap pages from the host preemptively only when we are confident that the device will access the pages before the host. A reasonable heuristic is to assume that if some portion of a data structure is accessed by the device, the full data allocation is likely to be accessed by the device. Under this reasoning, we perform unmapping using an event-driven methodology, where a page fault on a specific address triggers the unmapping of all memory within the chosen granularity.

Semantic Correctness. We need to ensure that the original semantics are preserved. In particular, we need to ensure data is unmapped from the host and evicted from host TLBs before data is migrated to the host. For sequential preemptive unmapping, this is not a concern. However, we also evaluate asynchronous methods of unmapping that could lead to race conditions.

Our method operates on VA Block-sized chunks within the chosen granularity, locking this region for mutual exclusion. If the main thread encounters a VA Block that has not yet been unmapped, it can fall back to the original implementation and unmap the code as needed; however, if it has been asynchronously mapped, we can skip this step and gain back our performance.

For the vast majority of codes, our described approach is effective. However, it does seem possible to program antagonistic scenarios. Mitigating these risks is important in taking this approach.

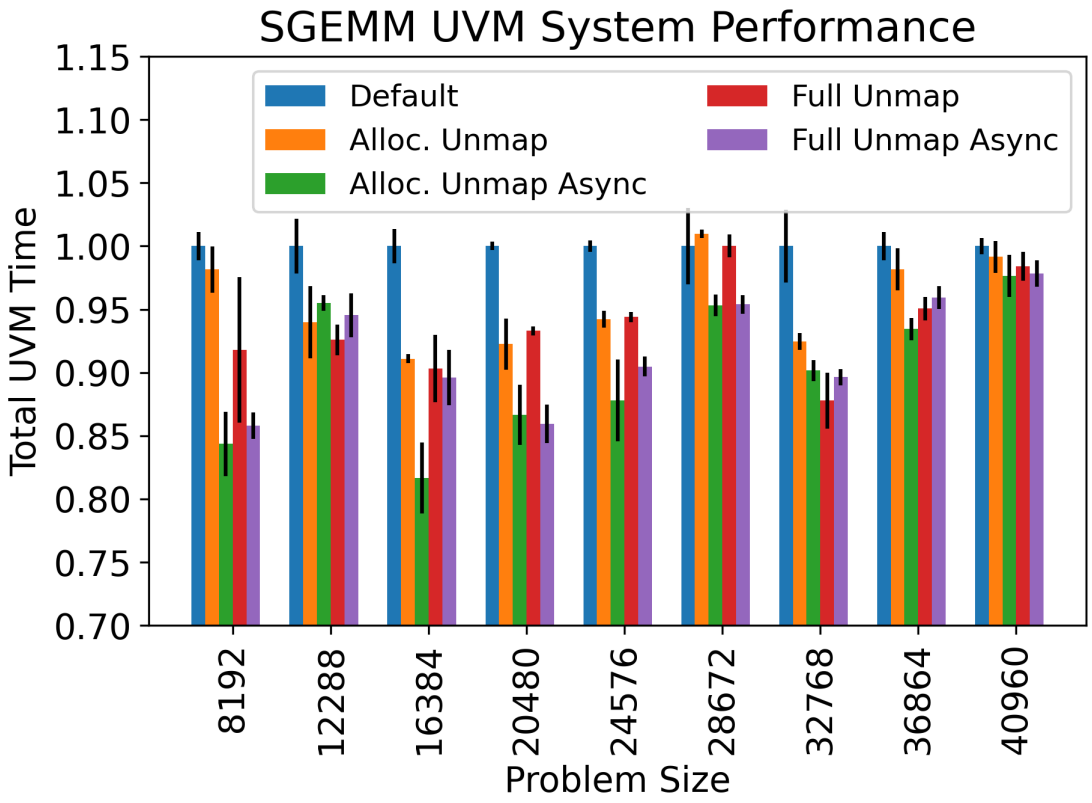


Figure 6.1: Total time spent in UVM execution for SGEMM kernels over increasing problem sizes for each driver configuration normalized to the default NVIDIA UVM configuration. Sizes above $30,000$ exceed GPU memory and use oversubscription. Preemptive unmapping generally improves performance, asynchronous preemptive unmapping always outperforms with potential for significant gains.

6.3 Evaluation of Preemptive Asynchronous Unmapping

In this section, we evaluate our proposed methodology and conditions against several benchmarks to evaluate its effectiveness at improving system performance, batch turnaround time, and end-to-end application performance. We revisit benchmarks from previous sections to analyze the benefits of our methods. We provide the standard error over five iterations to account for system variations. We also present our analyses of the impact of these improves as well as new bottlenecks in the UVM system.

6.3.1 System Performance

We demonstrate the overall system performance of the `sgemm` benchmark comparing our methods to the original implementation in figure 6.1. In this figure, we show that average system runtime is always reduced by asynchronous preemptive methods across problem sizes up to an 18.4% improvement. The benefit of this unmapping optimization is proportional to the size of data required by the kernel as all data must eventually be unmapped. Generally, the asynchronous full or allocation-based unmapping methods show the best performance.

Simply converting to preemptive unmapping can offer a minor improvement in performance; however, for applications where the full address space may not be used by the device, such as HPGMG or Tealeaf, this can have a detrimental impact on performance. Adding asynchronous unmapping mitigates this issue and provides performance benefits. Allocation-based asynchronous unmapping is shown to be more effective for end-to-end performance in this scenario to avoid unmapping data that the host will later use. Finally, benefits tend to trail off as oversubscription increases. In oversubscription scenarios, the magnitude of unmapping cost reduction continues to scale, but oversubscribed problems spend an increasing amount of system time performing eviction operations.

Figure 6.2 demonstrates that these results generalize by showing the optimal improvement across all applications in comparison to our asynchronous preemptive methods. The geometric mean improvement is $xx\%$, showing the method is widely applicable for improving system performance.

While the performance gains are substantial, our findings in prior sections indicate we can expect a higher speedup. Once again analyzing the batch performance breakdown, figure 6.3 shows that the *tracker wait time* increases for our improved methods. Within UVM, this is indicative of time spent waiting on data transfers or other device functions to execute. This indicates that by resolving our unmapping bottleneck, we have shifted the bottleneck to the interconnect. Future work will analyze how modern interconnects impact data transfers between hosts and devices.

6.3.2 End-to-End Performance

We present two microbenchmarks, `sgemm` and `stream` that show notable end-to-end application performance based on our implementation. In figure 6.4, we demonstrate that our asynchronous methods improve performance by over 8% with no modification of user code. This demonstrates that

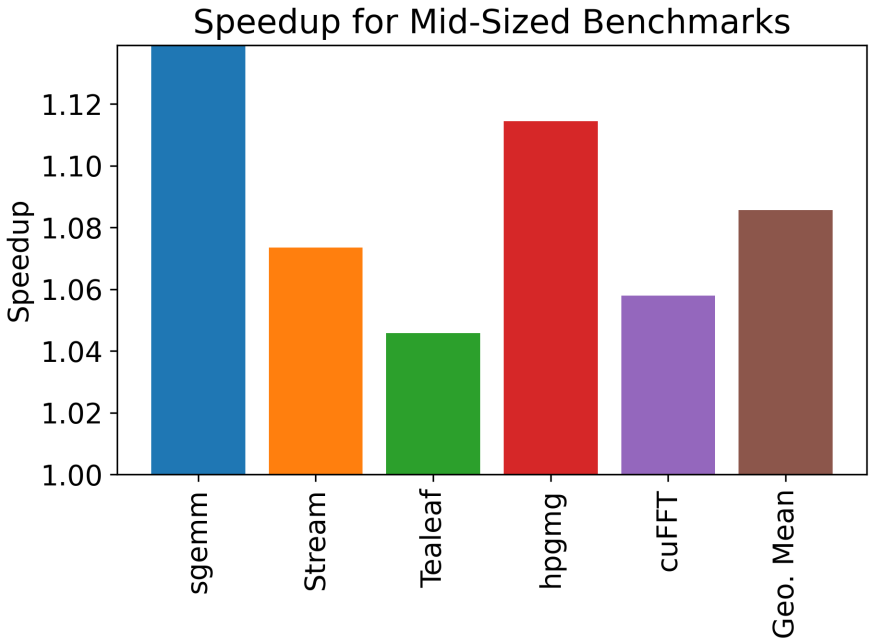


Figure 6.2: Performance for mid-to-high memory usage benchmark sizes on other applications. Performance improvement generalizes, although total system performance improvement will depend on the memory footprint and hybrid access pattern of individual applications as described in prior sections.

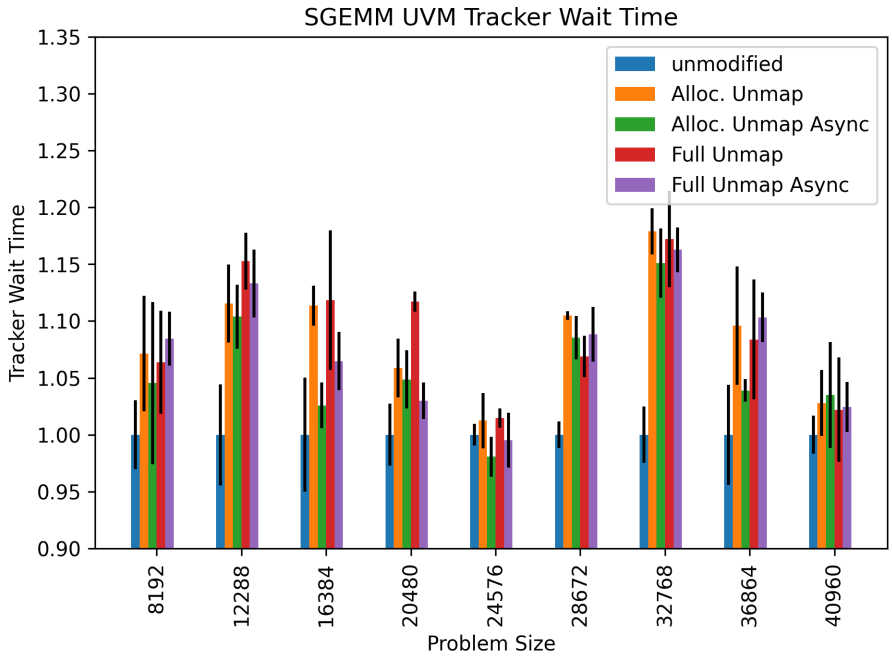


Figure 6.3: This figure shows the amount of time spent waiting to synchronize with the device. Our unmapping methods increase this as overall performance increases, likely indicating that we are shifting the bottleneck towards the device and interconnect.

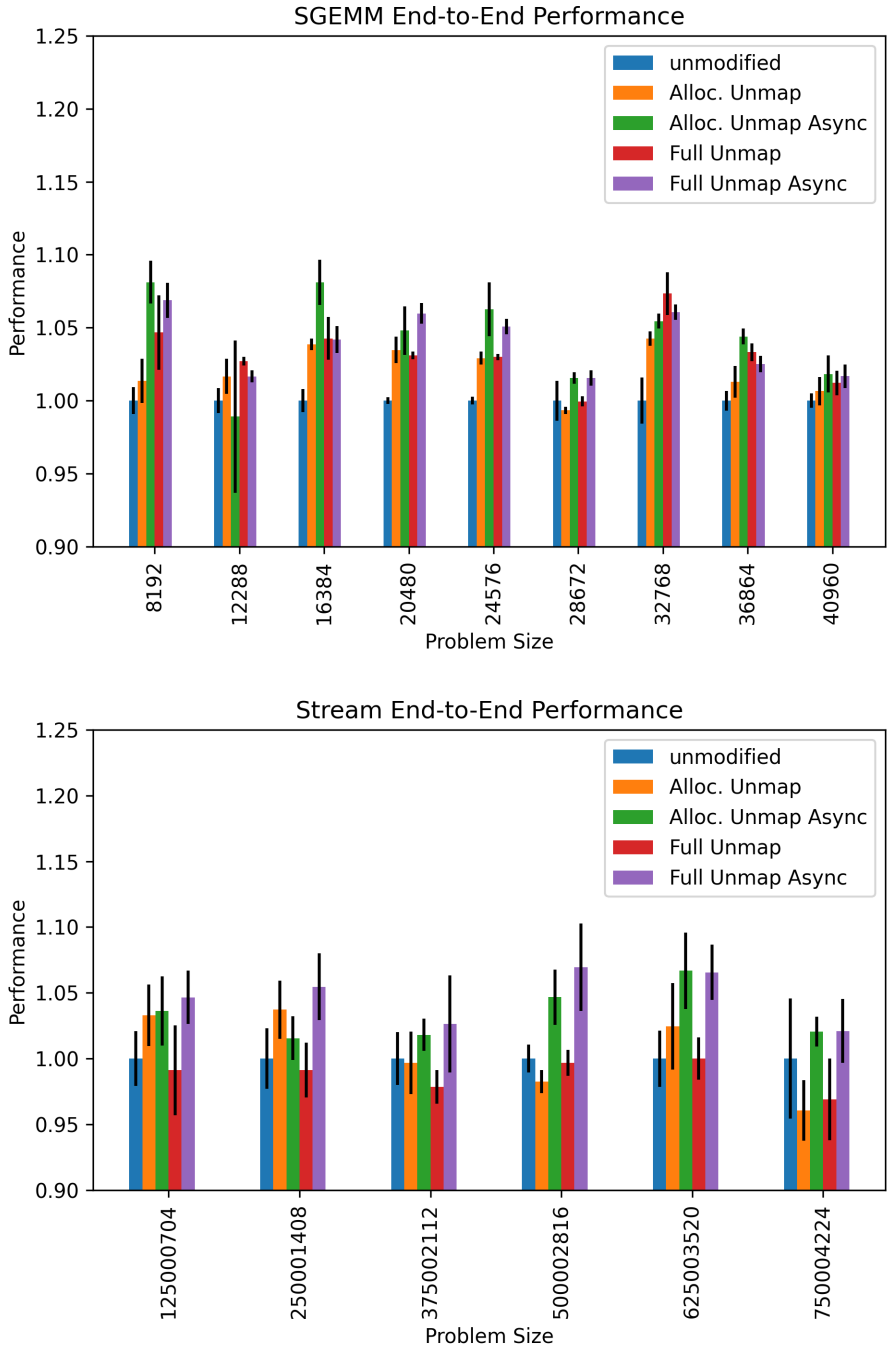


Figure 6.4: This figure shows the amount of time spent waiting to synchronize with the device. Our unmapping methods increase this as overall performance increases, likely indicating that we are shifting the bottleneck towards the device and interconnect.

for certain common use-cases, these optimizations can have significant performance improvement. The `sgemm` microbenchmark is a commonly used matrix multiplication computational kernel that can be used as a building block for other codes, such as deep neural networks. The results of these kernels are frequently required on the host for computing various iterative statistics, or for aggregating results across devices and nodes. In these circumstances, our changes could have significant impact on overall performance.

Chapter 7

Conclusion

In this work, we have analyzed and characterized the type of systems driving Unified Memory technologies for accelerator devices on modern systems. We have identified the key cost components with unexpected performance characteristics in the UVM fault path. We have examined these components with UVM-specific workload features, and highlight the impact of these features on overall performance and fault-batch workload processing. This work serves as an initial but thorough investigation into the systems-software performance and behavior concerns for UVM, as well as a proxy study for HMM and the interfacing vendors/devices in the future. Below we summarize the key findings and discuss them in a broader context as well as future required work.

7.1 Key Driver Costs and Optimization.

Data movement contributes only a small amount of overall cost in contrast to expectation, as most of the movement is asynchronous. With sufficient improvements to the underlying system software, this cost can become a much greater concern. This suggests that improvements of basic hardware, such as interconnect bandwidth and latency, will still improve performance but are not a replacement for improved migration algorithms.

Host OS operations, particularly unmapping CPU pages on the fault path, contribute significant overhead. These costs can be exacerbated by some user code parallelization schemes. We have traced these issues to come from the combined cost of page table management and TLB shootdowns. These issues can be mitigated with clever asynchronous management techniques in a method that

is portable to future systems. Other host costs, such as DMA setup and management, are also responsible for expensive systems operations and should be considered carefully in future systems to ensure consistent performance.

Driver Serialization.

Code generation and device-level throttling limit the generation of faults from each SM and ensure batches representing every SM on NVIDIA devices. Consequently, the GPU is generally stalled during driver fault processing, leading to highly synchronous behavior between the CPU and GPU with little overlap and high latency cost if there is not heavy data reuse on the GPU. This is the key reason driver performance is so important to overall performance.

The driver is a serial bottleneck for the parallel batch workloads created by the GPU. Ideally, this could be improved by parallelizing the driver. The current architecture would lend itself towards simple parallelization among VABlocks, but our workload analysis shows this would create a very imbalanced workload. Parallelizing faults per-SM may be more reasonable, if devices supported targeted per-SM replay. We have demonstrated the effectiveness of adding additional parallelism to fault handling, but significant software or hardware design alterations would need to be considered to create a balanced workload distribution for fault-handling. While these workload features are specific to NVIDIA GPUs, any vendor implementing HMM for parallel devices will encounter similar concerns and delays.

Prefetching and Eviction.

Density prefetching has moderate to high efficacy in non-oversubscribed workloads. Prefetching improves performance by eliminating a large number of batches. It also benefits from under-subscribed workloads by allowing the GPU to operate as an underfilled cache, demonstrating that more aggressive prefetching would be effective for these use cases. However, prefetching is unable to mitigate batches with high DMA and CPU unmapping overhead, increasing the impact of these costs in real workloads and sustaining a performance gap between direct-transfer workloads and UVM. We have also identified that density programming is difficult for users to program against, although it does have a moderate-to-high level of effectiveness.

Eviction, on the other hand, while necessary for large data sets, has cost similar to that of a page fault, but can also induce more overall page faults if done incorrectly. We demonstrate that

eviction creates tiered levels of performance per each VABlock evicted. To make oversubscription more cost-effective, it must be improved independently of host OS problems, as the underlying performance issues stem from algorithmic issues and user applications, not Host OS interference. However, the host OS can add onto eviction cost and increase the costs.

Prefetching and eviction have indirectly-related semantics that can magnify the costs of utilizing oversubscription. Prefetching can cause the movement of unneeded data to the GPU, creating increased data migration costs for no return on the investment. These two features must be developed to work together for future workloads.

7.2 Discussion and Future Work

After this complete study of performance, there are several avenues for ongoing work. First, there is a strong case to be made that the true potential of UVM is in effective oversubscription and prefetching strategies. If these two mechanisms can effectively interoperate with acceptable performance loss, it is an instant improvement over traditional direct management. Secondly, we have only studied UVM performance for single GPU systems. Independent multi-GPU systems are not significantly different, but cooperating peer GPUs can offer a very interesting path forward if implemented effectively. We discuss these two possibilities and their challenges here based on our prior findings.

7.2.1 Prefetching and Eviction for High Performance UVM

The key limiting factor for improved prefetching and eviction is a lack of readily-available information for making better decisions. We have discussed that the existing eviction mechanism uses a *least recently faulted* approach, and that prefetching is determined by density without acknowledging time. While this data is a useful heuristic for some use-cases, we have also demonstrated that it is insufficient for many others.

As a design consideration, additional information could be provided to hardware from software. For example, NVIDIA GPUs provide quite a bit of additional information along with a GPU fault, primarily for tracing higher-level information about the origin of a fault. This information is sufficient to trace the originating graphics processing cluster (GPC) and perhaps the specific µTLB that generated the fault. Another level of information that offers SM ID, logical thread ID, or related

information sufficient to pinpoint a specific area of execution, as well as program counters for source code correlation, could open the door for existing prefetching methods from literature at the cost of additional fault buffer overhead. Prefetcher technology in hardware has been explored deeply for CPU and GPU technology; if a degree of greater information was available, then software could explore these techniques at a wider scale without needing to devote substantial hardware to built-in hardware prefetching.

In the absence of additional information, we can consider further exploring GPU access patterns. NVIDIA has included support for multiple-granularity access counters for GPU-level memory access on GPUs since the Volta architecture [34]. Access counters are only capable of providing information on remote memory accesses, which we do not explore in this work. However, they could be leveraged to provide more detailed access pattern information on GPUs without relying on simulators. We can use these to better explore how access patterns and locality occur on GPU systems.

Another possibility is to use access counters directly. Presently, access counters can be used to enable a delayed-migration scheme, where data will be accessed remotely until a certain density and then migrated. Some further expansion of this idea is explored and simulated in Ganguly et al.[16], but has not been explored on a real system. This information opens the door to many types of adaptive eviction and prefetching and requires further study alongside the study of access patterns.

7.2.2 Peer-to-Peer GPU Programming

Peer-to-Peer GPU communication using UVM is an unexplored area. While multi-GPU systems are popular, the predominating programming model is to use independent MPI processes to drive independent GPUs. By allowing GPUs to talk directly and request data from each other, it potentially opens the door to much more flexible programming models that could offer higher performance in cases where the host can be eliminated as party to memory migrations. The primary challenges with this are the added complexity in programming for memory safety, and the performance is not well understood enough for experimentation. Our analysis techniques can extend into the Peer-to-Peer domain to better understand how this process works and if this method is fast enough on modern interconnects for effective use. This extends beyond the scope of NVIDIA GPUs, and could be use to create collaborative computation with other accelerators, such as target ASICs

or FPGAs.

Bibliography

[1] Top500. https://www.top500.org/. Accessed: 2021-06-15.

[2] Amogh Akshintala, Vance Miller, Donald E. Porter, and Christopher J. Rossbach. Talk to my neighbors transport: Decentralized data transfer and scheduling among accelerators. *Proceedings of the 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*.

[3] Tyler Allen and Rong Ge. Demystifying gpu uvm cost with deep runtime and workload analysis. In *35th IEEE International Parallel and Distributed Processing Symposium*. IEEE, May 2021.

[4] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 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, page 136–150, New York, NY, USA, 2017. Association for Computing Machinery.

[5] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli. Griffin: Hardware-software support for efficient page migration in multi-gpu systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 596–609, 2020.

[6] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland. Raja: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, 2019.

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

[8] Chia-Hao Chang, Adithya Kumar, and Anand Sivasubramaniam. 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*, SYSTOR '21, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[10] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*. Wrox Press Ltd., GBR, 1st edition, 2014.

[11] Steven Chien, Ivy Peng, and Stefano Markidis. Performance evaluation of advanced features in cuda unified memory. *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, Nov 2019.

[12] Linux Kernel Development Community. Heterogeneous memory management (hmm). https://www.kernel.org/doc/html/latest/vm/hmm.html. Accessed: 2021-06-15.

[13] UK Mini-App Consortium. Tealeaf. "`https://github.com/UK-MAC/TeaLeaf_CUDA`", 2016. Accessed: 2021-06-15.

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

[15] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 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, pages 224–235, New York, NY, USA, 2019. ACM.

[16] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, may 2020.

[17] R. Gayatri, K. Gott, and J. Deslippe. 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)*, pages 41–46, 2019.

[18] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. Traversing large graphs on gpus with unified memory. *Proc. VLDB Endow.*, 13(7):1119–1133, March 2020.

[19] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus, 2020. arXiv:2007.09822.

[20] Dion Harris. Green light! top500 speeds up, saves energy with nvidia. https://blogs.nvidia.com/blog/2020/06/22/top500-isc-supercomputing/. Accessed: 2021-06-15.

[21] Y. He, S. Wan, N. Xiong, and J. H. Park. A new prefetching strategy based on access density in linux. In *International Symposium on Computer Science and its Applications*, pages 22–27, 2008.

[22] 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. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, September 2005.

[23] John Hubbard and Jerome Glisee. Gpus: Hmm: Heterogeneous memory management. https://www.redhat.com/files/summit/session-assets/2017/S104078-hubbard.pdf, May 4, 2017. Accessed: 2021-06-15.

[24] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 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*. ACM, mar 2020.

[25] Marcin Knap and Paweł Czarnul. Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus. *The Journal of Supercomputing*, 75:7625–7645, November 2019.

[26] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014.

[27] Pak Markthub, M. E. Belviranli, S. Lee, J. Vetter, and S. Matsuoka. Dragon: Breaking gpu memory capacity limits with direct nvm access. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426, 2018.

[28] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, October 2020.

[29] Saiful A. Mojumder, Yifan Sun, Leila Delshadtehrani, Yenai Ma, Trinayan Baruah, José L. Abellán, John Kim, David Kaeli, and Ajay Joshi. Mgpu-tsm: A multi-gpu system with truly shared memory, 2020. arxiv:2008.02300.

[30] J. M. Nadal-Serrano and M. Lopez-Vallejo. 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):1579–1588, 2016.

[31] NVIDIA. cublas. https://developer.nvidia.com/cublas. Accessed: 2021-06-15.

[32] NVIDIA. cufft. Accessed: 2021-06-15.

[33] NVIDIA. Nvidia a100 tensor core gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. Accessed: 2021-06-15.

[34] NVIDIA. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 2021-06-15.

[35] NVIDIA. Open gpu documentation. https://nvidia.github.io/open-gpu-doc/. Accessed: 2021-06-15.

[36] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[37] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: Minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Nikolay Sakharnykh. High-performance geometric multi-grid with gpu acceleration. https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/, feb 2016. Accessed: 2021-06-15.

[39] Nikolay Sakharnykh. Memory management on modern gpu architectures. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9727-memory-management-on-modern-gpu-architectures.pdf, May 19, 2019. Accessed: 2021-06-15.

[40] IBM POWER9 NPU team. Functionality and performance of nvlink with ibm power9 processors. *IBM Journal of Research and Development*, 62(4/5):9:1–9:10, 2018.

[41] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *ACM Trans. Archit. Code Optim.*, 18(2), February 2021.

[42] Hailu Xu, Murali Emani, Pei-Hung Lin, Liting Hu, and Chunhua Liao. Machine learning guided optimal use of gpu unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 64–70, 2019.

[43] Q. Yu, B. Childers, L. Huang, C. Qian, H. Guo, and Z. Wang. Coordinated page prefetch and eviction for memory oversubscription management in gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 472–482, 2020.

[44] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. A quantitative evaluation of unified memory in GPUs. *The Journal of Supercomputing*, 76(4):2958–2985, nov 2019.

[45] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357, 2016.