In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing

Tyler Allen
School of Computing
Clemson University
tnallen@clemson.edu

Rong Ge
School of Computing
Clemson University
rge@clemson.edu

ABSTRACT

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

In this paper, we take a deep dive into the UVM system architecture and the internal behaviors of page fault generation and servicing. We reveal 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 implementations, warranting close attention. This study serves as a proxy for future shared memory systems such as those that interface with HMM.

KEYWORDS

UVM, NVIDIA, GPU, virtual memory, GPGPU, HMM

ACM Reference Format:
Tyler Allen and Rong Ge. 2021. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3480855

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3480855

1 INTRODUCTION

Graphics Processing Units (GPUs) have become a computational mainstay in modern HPC systems and are paving the way for other accelerators into the HPC space. Natively, discrete GPUs have separate physical memory traditionally programmed through API and managed by device drivers. Multiple technologies that ease the burden of programming and increase codebase portability with these accelerators by abstracting the complexity of separate CPU and GPU physical memory domains are under ongoing development. *Heterogeneous Memory Management* (HMM) and NVIDIA Unified Virtual Memory (UVM) are two such independent yet potentially collaborative efforts. These technologies integrate device memory domains into the OS virtual memory system and transparently migrate pages across devices. HMM is a Linux kernel feature that provides a generic interface for heterogeneous memory management to vendor- and device-specific drivers on commodity systems [11, 20]. NVIDIA UVM presently offers an all-in-one approach combining paging and device drivers for NVIDIA GPUs. It can also integrate with the HMM interface [33]. As of today, NVIDIA UVM alone has been prolific, adopted by the US Department of Energy and in common HPC frameworks such as Raja [6], Kokkos [8], and Trilinos [19].

As noted by prior studies, transparent paging and migration come with heavy performance costs [2, 18, 21–23, 37]. Figure 1 shows that the access latency generally increases one or more orders of magnitude compared to explicit direct management by programmers. While such costs may be acceptable for applications computing in-core on GPU memory, high-performance systems suffer inefficient utilization as a consequence. Further, the out-of-core capability comes at a much greater cost, largely prohibitive for most applications. Prefetching mitigates but cannot overcome all of the cost and could prohibitively increase it for some memory-oversubscribed workloads [2, 14, 16, 22, 36, 38].

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

In this work, we take a deep dive into the UVM system architecture and the internal behaviors of page fault generation and

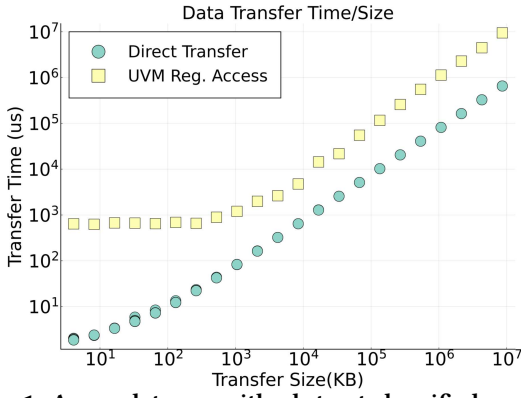


Figure 1: Access latency with abstracted unified space increases by one or more orders of magnitude over explicit direct management.

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

Our work examines the real-time functionality of the system on real hardware. We provide a deep understanding of the interaction between CPU and GPU with UVM and the costs of different functionalities. In particular, we make the following contributions:

- We conduct an in-depth study of GPU page fault generation and how UVM aggregates faults into fault batches - the core UVM work unit - to understand the UVM workload better.
- We take a closer look at how UVM serves page faults within a batch through the page fault handling path, offering perspective and rationale behind design decisions and constraints.
- We analyze UVM as an example of future HMM systems, isolating performance considerations to vendor-specific and common code between all implementations and discussing improvements and considerations for different cases.
- While this work focuses on single GPUs, it serves as a base and foundation for studying the interactions among multiple devices on the same systems, which are the standard building blocks of computer clusters.

2 UVM BACKGROUND AND RELATED WORK

NVIDIA UVM has the same functional philosophy as the likely future industry standard, HMM — Linux-like virtual memory through paging, where *page faults* trigger *data migration* between the host memory and accelerators [11, 20]. From the programmer's perspective, HMM is preferable as it allows the same memory management functions as for the CPUs, whereas UVM requires special memory allocation functions to achieve the same result. However, HMM requires backend device-specific solutions from vendors [11, 20]. **The NVIDIA UVM driver is among the first backend solutions to interface with HMM. However, to the best of our knowledge, the full integration for x86 systems is not available yet [33].**

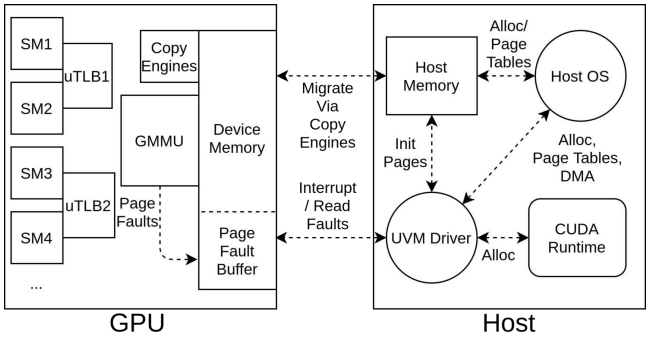


Figure 2: The UVM architecture. The UVM driver resides on the host and manages the fault buffer on the device.

NVIDIA UVM is currently the primary real-world implementation of transparent paging and migration across memory domains. Thus we focus on UVM but draw insights applicable to HMM.

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

2.1 The UVM Architecture

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

Any active thread on the GPU can trigger a page fault. The page fault is recognized and handled by the hardware thread's corresponding µTLB [29]. The thread treats this scenario as any other outstanding memory request and may continue executing instructions not blocked by a memory dependency. Meanwhile, the fault propagates to the GPU memory management unit (GMMU), which sends a hardware interrupt to the host. The GMMU writes the corresponding fault information into the *GPU Fault Buffer*. The fault buffer acts as a circular array, configured and managed by the UVM driver [29]. The `nvidia-uvm` driver fetches the fault information, caches it on the host, and services the faults through page processing and migration.

The GPU exposes two functionalities to the host via the GPU command push-buffer — host-to-GPU memory copy and fault replay. As part of the fault servicing process, the driver instructs the GPU to copy pages into its memory, generally using high-performance hardware "copy engines." Once the GPU's page tables are updated and the data is successfully migrated, the driver issues a fault replay [39], which clears the waiting status of µTLB, causing them to "replay" the prior miss.

2.2 Fault Batching and Handling

The `nvidia-uvm` driver groups outstanding faults into **batches** in the host-side cache. Fault delivery to the host requires two steps: first, the GPU sends an interrupt over the interconnect to alert the host UVM driver of a page fault, and second, the host retrieves the

complete fault information from the *GPU Fault Buffer*. The interrupt wakes up a worker thread to begin fault servicing if none are awake. UVM uses batching as an optimization as it allows the driver to ignore most interrupts. The default fault retrieval policy is to read faults until the batch size limit is reached or no faults remain in the buffer. Batches contain up to a maximum size of 256 faults. The worker thread tries to service another after one batch and sleeps if it finds no new faults. For comparison, device drivers are still responsible for these actions in HMM implementations. Fault batching and fault handling policies are the driver's independent decisions.

For compatibility with the host OS and future HMM implementations, UVM adopts the host OS's page size for migration and tracking: 4KB pages for x86 systems and 64KB pages for Power9 systems. UVM has additional internal abstraction for management and performance considerations. For x86, pages are upgraded from 4KB to 64KB within the UVM runtime *as a component of prefetching*, emulating the 64KB Power9 page size. Additionally, the driver splits all memory allocations into 2MB logical Virtual Address Blocks (VABlocks). These VABlocks serve as logical boundaries; the driver processes all batch faults within a single VABlock together, and each VABlock within a batch requires a distinct processing step. UVM also tracks all physical GPU memory allocations from the `nvidia` resource manager. If eviction is required, UVM evicts allocations at the VABlock granularity.

2.3 Related Work

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

High-Level Analysis and Application Optimization. High-level analysis typically focuses on either comparing UVM to traditional manually-managed memory applications or comparing UVM across different hardware platforms such as Power9 vs. x86_64 and NVLINK vs. PCIe. The overall performance impact of UVM was studied in [22, 23, 37] on several applications for both non-oversubscription and oversubscription. Manian et al. study UVM performance and its cooperation with MPI across several MPI implementations [25]. Gu et al. produce a suite of benchmarks based on the Rodinia benchmark suite to perform these kinds of evaluations [18]. Markidis et al. focus on advanced features of UVM, such as runtime allocation hints and properties [10], while Gayatri et al. focus on the impacts of prefetching and Power9 Address Translation Services (ATS) [16]. Several works have tried to improve graph-processing or graph-specific applications that have known irregular processing by utilizing the remote mapping (DMA) capabilities of UVM as well as altering access patterns or data ordering to make accesses less irregular [17, 26, 28].

Hardware and System Alterations. Some works discuss fundamental changes to the UVM architecture or UVM hardware to improve overall performance, whereas our work focuses on identifying performance characteristics and issues that are solvable on existing hardware/software. Griffin offers architectural changes to enhance page locality for multi-GPU systems [4]. Kim et al. simulate "virtual threads" to effectively increase the overall number of threads resident on the GPU to better hide latency, along with increasing the fault batch size to allow the host to process more faults at the same time [21]. Several works suggest replacements for UVM that diverge from the demand-paging paradigm [3, 27]. Ganguly et al. use the existing but sparsely utilized page counters system within the existing UVM ecosystem to improve performance for memory-oversubscribed workloads [15] and offer modifications to eviction and prefetching algorithms after integrating these features into hardware [14]. Similarly, Yu et al. also offer architectural changes to coordinate eviction and prefetching [36].

UVM System Analysis. These works are the most similar to ours. Allen and Ge focus on the driver-level performance of prefetching, showing page-level access patterns and performance data for the general case, but not the root source of UVM costs [2]. Kim et al. show an example of batch-level size/performance data similar to ours [21]. In contrast, our work dives into the software and hardware-based root causes under different scenarios and analyzes the construction of these batches.

3 UVM FAULT BEHAVIORS

In this section, we focus on revealing the behavior of faults generated on the GPU. In particular, we demonstrate the following:

- 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 require additional page faults.

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.

3.1 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.

We collect all data through a modified UVM driver distributed alongside the NVIDIA driver. We modify the UVM driver into two versions. One logs per-fault metadata for gathering overall statistics about faults such as their GPU SM of origin. The other is instrumented with targeted high-precision timers and event counters for collecting batch-level data. Batch data is logged to the system log at the end of each batch. We use a custom logging tool that is more reliable than `dmesg`.

For data presented in this work, we use the applications in table 1. They are representative HPC applications, i.e., the kernels including `sgemm`, Gauss-Seidel, and FFT are commonly used in various HPC applications, and HPGMG is a full proxy application representing algebraic multigrid methods.

Table 1: Benchmarks used in evaluation and analysis.

Benchmark	HPC Use Examples
cuBLAS sgemm	Fluid Dynamics [34], Finite Element [5], Deep Learning [9]
stream	Memory bandwidth (triad-only) [12]
cuFFT	LAMMPs [30, 35], Particle Apps [31], Molecular Dynamics [35], Deep Learning [24]
Gauss-Seidel	HPCG [13], AMR [7]
HPGMG-FV	Proxy App for AMR [1]

3.2 Formation of GPU Fault Batches

In UVM, the **fault batch** is the fundamental unit of work. 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 [2, 21]. 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 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 one 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 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 2: Annotated SASS assembly corresponding to line 8 in listing 1.

```
--- snip ---
LDG.E.SYS R9, [R2] ; <-- a[page0]
LDG.E.SYS R0, [R4] ; <-- b[page0]
FADD R9, R0, R9 ; <-- scoreboard stalls: 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 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 vector A reads and most vector B reads.

We draw two insights from this first batch of reads. (1) Each *thread* can perform one or more memory read instructions resulting in faults without blocking, the exact 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

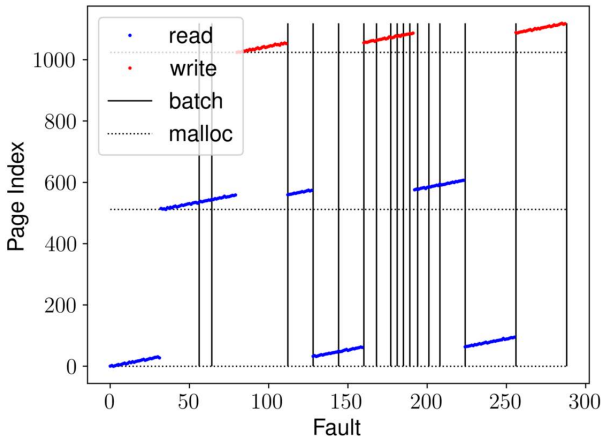


Figure 3: Faults of vector addition as a relative time series.

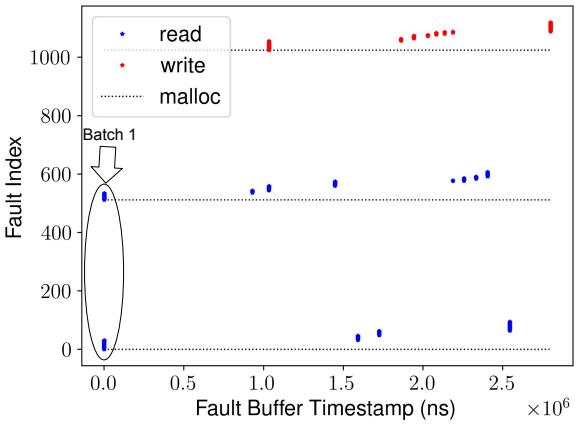


Figure 4: Faults of vector addition with real-time timestamps of arrival to the fault buffer. Faults clustered tightly vertically always indicate a batch.

comparing against larger and more complex examples. Figure 4 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.

We observe a subtle faulting behavior from the second and third batch of Figures 3 and 4: no write accesses can execute until all 64 prerequisite reads have been fulfilled, even though the required memory addresses are known upfront. This behavior is traceable to a subtle but consistent coding practice demonstrated in the resultant SASS assembly code in Listing 2 for one iteration of the vector addition. It becomes clear that the intermediate result of $A + B$ is required before the result can be stored in vector C and the corresponding page fault is generated. A coalescing version of the vector addition code 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 infer that in addition to the µTLB fault limit, there is an additional fault rate throttling mechanism prevents a single SM from creating too many faults. In Figure 3, several batches consist of a small number (<< 56) of faults, even

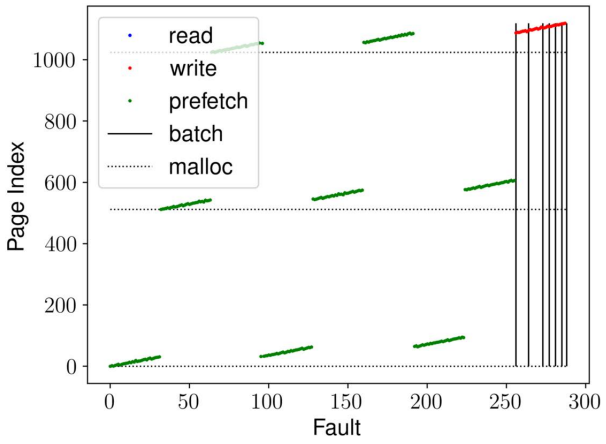


Figure 5: A single warp can generate faults up to the batch size limit using prefetching.

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 [39].

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, a page fault is triggered if the data is not present in global memory. Prefetching is unique because it does not require the register scoreboard, thus presumably avoiding triggering the previously-mentioned limitations. Figure 5 shows the resulting batches, where vectors A, B, and C are prefetched upfront. A single warp can generate up to 256 faults in a single batch, capped by the software batch size limit [1]. This behavior far exceeds the prior per-SM fault generation capabilities, confirming our prior assertions about code limitations fault-throttling.

Table 2: Per-SM Source Statistics in Each Batch

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
gauss-seidel	0.65	0.45	0.01	2.95
hpgmg	0.41	0.10	0.01	2.65

In table 2, 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 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 behaviors discussed previously, and it shows that SMs are served relatively "fairly."

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

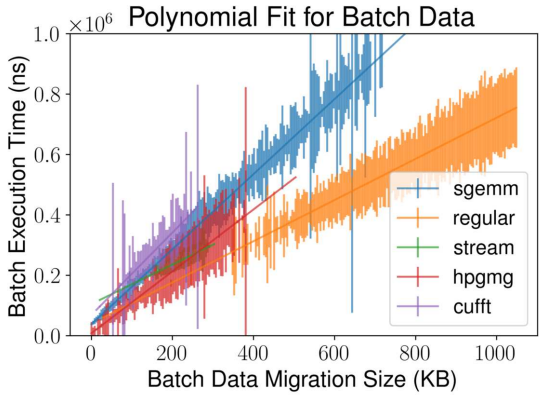


Figure 6: Best fit of batch sizes vs. data migrated for one run of several applications.

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 place between data accesses. In the next section, we look at how characteristics of the generated workloads influence overall performance.

4 UVM DRIVER WORKLOAD

The UVM fault batching shapes the resulting UVM driver workload, and the overall performance is determined by how the driver handles 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 appropriate handling. Interestingly, some applications fit both categories at different points in a single kernel, creating a complex and difficult-to-optimize scenario for the driver. We investigate several key 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 determines the trend for performance variance.
- Host OS interaction: some components, such as CPU page unmapping, require the host OS and surprisingly incur significant overhead on the fault path.

This section explores how these characteristics influence batch performance and, in turn, overall application performance.

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 6 demonstrates that the average batch cost rises linearly with the amount of data moved for all applications. However, the average cost differs with applications, and there is a high variance for each application.

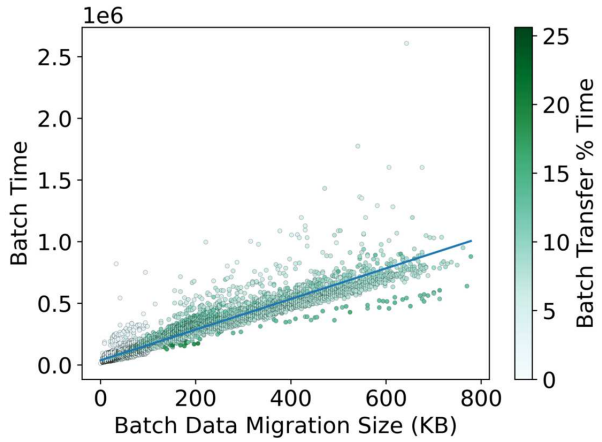


Figure 7: The percentage of time spent per batch performing data transfer for `sgemm`. At most, the transfer time is approximately 25% of the total batch time but is typically far lower.

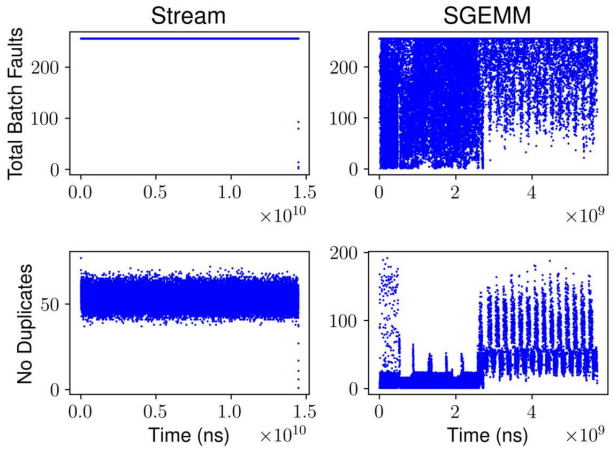


Figure 8: Batches in time series for stream and `sgemm`. Upper: fault counts registered by the driver. Lower: fault counts with duplicate faults to the same pages removed.

Even though data movement is a crucial 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 7 shows that transfer time accounts for less than 25% of the total batch time for almost all batches. This observation offers two insights: (1) Most batch servicing time is *not* spent on data transfer. While faster hardware may benefit performance, the more significant issue is ensuring the driver efficiently utilizes the interconnect subsystem. (2) The variance and skew must be derived from batch characteristics, the driver software, and the driver's interaction with the host OS and hardware. We investigate the constituent components of the overall performance cost, including variance, in the remainder of this section.

4.2 Duplicate Faults vs. Batch Size

To further understand the characteristics of batches over an application's lifetime and examine the causes of variance, we examine the impact of *duplicate faults* on the overall batch performance. We demonstrate that (1) the UVM driver workload is not uniform across applications and non-trivial benchmarks have varying batch characteristics, and (2) performance is more complex than just faults per batch with duplicate faults as one factor.

First, we demonstrate that the UVM workload is application-driven in terms of size and the number of duplicate faults. Figure 8 shows the actual batch size of all batches in an application execution as a time series, where the upper pair presents the raw batch size as pulled from the GPU fault buffer and the lower pair shows the number of faults in each batch after duplicate faults have been discarded. `sgemm` is far more complex than `stream` in implementation, and such complexity manifests in the changes and "phases" of the batching behavior over time. Filtering out duplicates greatly alters the average batch size for both applications, indicating the need to address duplicate faults. However, the impact of duplicates is not the same across applications or even within the same application. In the context of batch workloads, such non-uniformity explains portions of the variation in batch distribution previously seen in Figure 6, as duplicate faults do not contribute to the migration size but certainly account for a portion of overhead.

The driver classifies duplicate faults into two types: (1) faults to the same address that originate from the same µTLB, and (2) faults to the same address that originate from different µTLBs. The driver handles these types at different times, and the latter has a greater cost. Faults of type (1) commonly occur in codes with high spatial locality within a warp or block, causing multiple threads to issue the same fault; our data also indicate that SMs spuriously wake up to reissue the same fault during a batch. Type (2) duplicates indicate that there is data sharing among different blocks, and as such, some type (2) faults fall into type (1) because adjacent SMs share a µTLB. The reason for this distinction seems to be for more detailed metadata tracking and potential future improvements, and the difference is essential when considering alterations to how the GPU handles duplicate faults. However, for the data presented, we combine these types of duplicates as other costs overtake it.

Between batches, the fault buffer is flushed before a fault replay; any outstanding faults are dropped, and only faults that still need to be serviced will be reissued. The flush allows large numbers of duplicates to be dropped to reduce bulk transfers at the expense of overhead for dropping non-duplicate faults. We investigate this tradeoff by comparing the performance of various batch sizes. Figure 9 shows the results with the default batch size of 256. Critically, performance is generally greater with larger batch sizes, even though larger batch sizes have higher rates of duplicate faults. As larger batch sizes lead to smaller numbers of batches for the same problem size, we derive that the overhead of performing a batch is more costly than processing a modest number of 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. The number

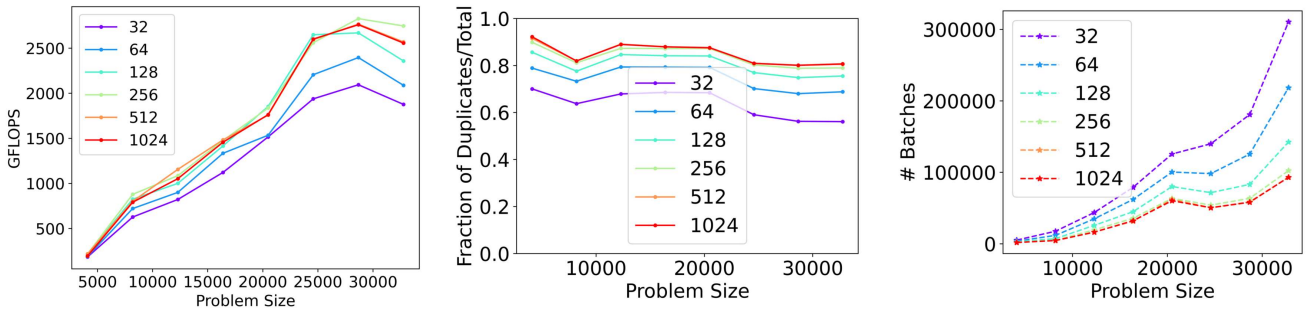


Figure 9: Batch size evaluation example with `sgemm`. There is a strong correlation between performance and batch size. Batch sizes up to 6144 (max) are tested but are not shown as performance does not change.

of total faults available per batch is limited by 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 next examine the distribution of faults in a batch across memory, e.g. spatial locality at the page granularity. Within the UVM driver, all operations are logically separated on VABlock (page-aligned 2MB) regions, making VABlocks a source of performance variation. Figure 10 shows batches colored by the number of unique VABlocks present in the data transfers for each batch. While each batch is subject to other sources of variance, one major trend is that, for batches with similar workloads, more VABlocks incur higher costs and cause more significant performance variation. This behavior is consistent with our earlier observation that the driver handles VABlocks within a batch independently.

Table 3: VABlock Source Statistics in a Batch.

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

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

4.4 Host OS Interaction

Management operations for host memory frequently require expensive interactions with the host OS. The host component of UVM is built on top of the existing virtual memory system in the Linux kernel. Because of this, migrations are subject to additional latencies incurred by existing mappings and the underlying virtual memory subsystem. We use an existing, UVM-optimized application to demonstrate this issue - the HPGMG implementation provided by NVIDIA [32]. Figure 11 shows an example of CPU-side behavior influencing GPU-fault performance outcomes. The two subfigures show the same problem with the same configuration, except (a) uses a single OpenMP thread, whereas (b) uses the default OpenMP thread configuration (one thread per logical core). Notably, the former configuration shows roughly twice the performance by simply disabling multithreading, and the performance trend falls in line with other applications that we have seen for a given data size.

Further, page unmapping represents a significant portion of execution time for many batches, as represented by the tone of color in Figure 11. Page unmapping is an operation in the existing virtual memory system on the host that UVM extends to support faults from GPU. Page unmapping is performed when the GPU touches a VABlock that is partially resident on the CPU. In this scenario, the driver calls into the kernel function `unmap_mapping_range()` to unmap all pages within the VABlock that are resident in host memory as part of the page migration. Interestingly, we observe that OpenMP multithreading exaggerates this specific cost for HPGMG. We note that this behavior does not occur in trivial cases, such as parallelizing data initialization in the `sgemm` application, indicating that data access patterns and thread affinity play a role in this issue.

We draw two conclusions about host OS interaction from the data presented: (1) unmapping host-side data takes place on the fault path and incurs significant overhead, and (2) certain host-side parallelizations of an application using UVM can exaggerate these unmapping costs. The host OS performs this operation, and the costs likely stem from issues with virtual mappings across CPU cores, flushing dirty pages from caches and TLBs, NUMA, and other memory-adjacent issues. Additionally, these operations do not take place in bulk due to the logical separation of VABlocks within UVM. This is an area that deserves particular scrutiny as HMM also performs host page unmapping on the fault path using host OS mechanisms, implying a similar cost could be applied to all devices when using HMM [11, 20]. Design and implementation issues such as how unmapping takes place and if it needs to be performed on-demand deserve further investigation.

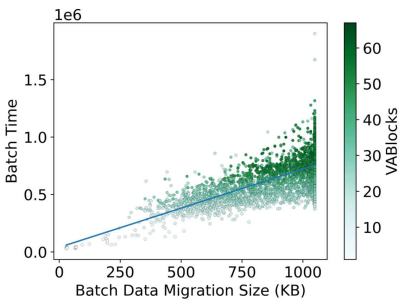


Figure 10: Batch time vs. to-GPU data migration size. For the same size, a higher cost is associated with more VABlocks.

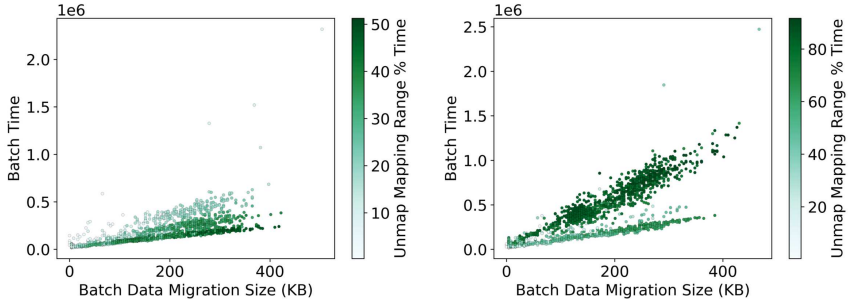


Figure 11: HPGMG with single threading (a) and multithreading (b), where the percentage indicates relative time per batch spent unmapping host-resident pages. Multithreading incurs larger percentages for unmapping.

5 WORKLOADS WITH PREFETCHING AND OVERSUBSCRIPTION

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

5.1 Oversubscription

Oversubscription allows applications to exceed GPU memory capacity by using a form of LRU eviction to swap pages back to the host. When all GPU memory has been previously allocated, eviction automatically migrates "cold" data back to the host to make room for new data at the granularity of 2MB VABlock. Figure 12 shows batch timing data for `sgemm` using a problem size that exceeds GPU memory. The application follows a somewhat expected trend in terms of batch distribution: many batches are executed before full GPU memory allocation without requiring eviction, and others (colored) evict one or more VABlocks. Predictably, blocks containing evictions incur greater overheads to (1) fail allocation, (2) evict a VABlock and migrate the data back to the host, and (3) restart the block migration process, including *page population*, a process by which pages are filled with zero values before data is migrated to them.

In Figure 13, 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 on the GPU but is later evicted, then it is not remapped to the CPU unless the CPU accesses it. 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.

5.2 Prefetching

UVM utilizes a runtime prefetching routine as part of the default behavior. The prefetching mechanism is a type of *density prefetching*, sometimes called *tree-based prefetching*, and is described in detail in [2, 14, 21]. The prefetcher's scope is limited to within a single VABlock and is only reactive; the prefetcher only flags pages within a VABlock currently being serviced for faults up to the full VABlock.

In Figure 14, we see the results of prefetching on the previously-viewed applications. The number of batches is reduced by 93% from the previous Figure 7 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 14, 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. Inline 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. However, we do not present this data here as the low-level timing creates significant skew in the overall timing information.

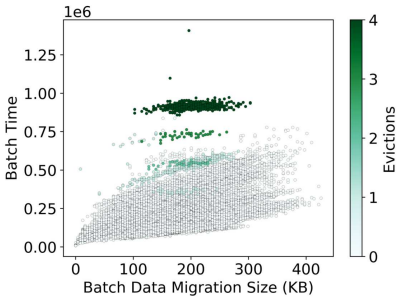


Figure 12: sgemm under oversubscription and eviction.

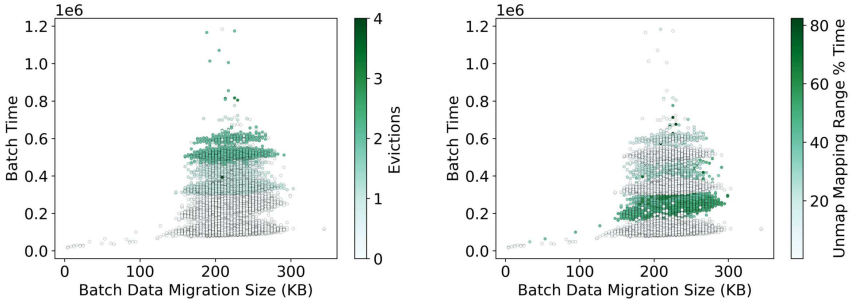


Figure 13: Stream under oversubscription. Left: multiple "levels" for the same eviction count. Right: a level may not include a portion of CPU unmapping.

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 more significant proportion of the overall cost.

5.3 Eviction + Prefetching

Finally, eviction combining prefetching creates the most complex scenario. Prior work has shown that the combination of prefetching and eviction can harm performance for applications with irregular access patterns [2, 21, 37]. The relationship is somewhat indirect since prefetching contained within a resident 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 subsequent eviction. We evaluate this scenario by comparing prefetching enabled and disabled scenarios for the same applications.

Figure 15 shows dgemm with combined eviction and prefetching properties in the migration size-sorted plot and as a time series. The range of data transfers is still extended but not to the full 20MB range observed in the prefetching example alone; we attribute this to reduced block access density for the larger problem size.

We examine each pair of figures individually: (1) In Figure 15a, we 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 robust prefetching response. (2) Figure 15b shows eviction ranges remarkably similar to the non-prefetching data set, fitting into the same sizes and ranges. The eviction set has relatively low batch sizes because evictions are caused by paging in *new* VABlocks, which have low access density at first. (3) In Figure 15c, non-eviction batches that include new VABlocks tend to have smaller 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 15d, we observe that creating DMA mappings can still have high overhead, although it is intermittent. This figure suggests that

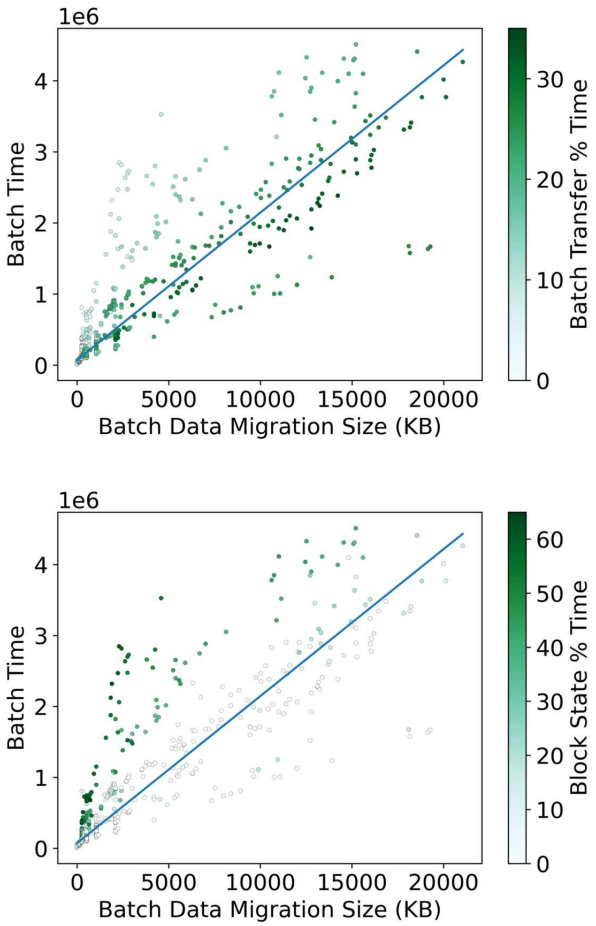
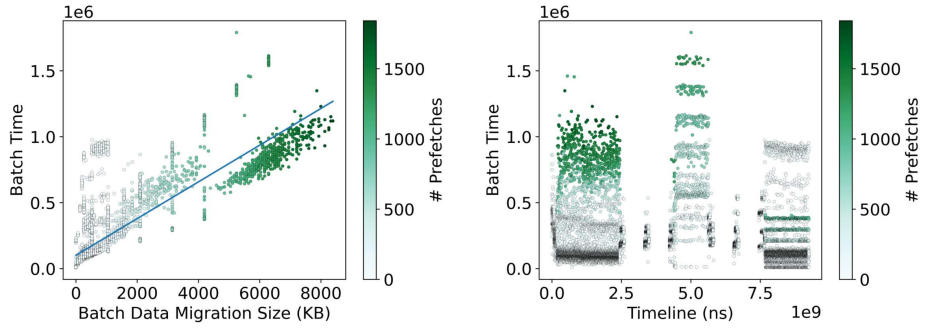


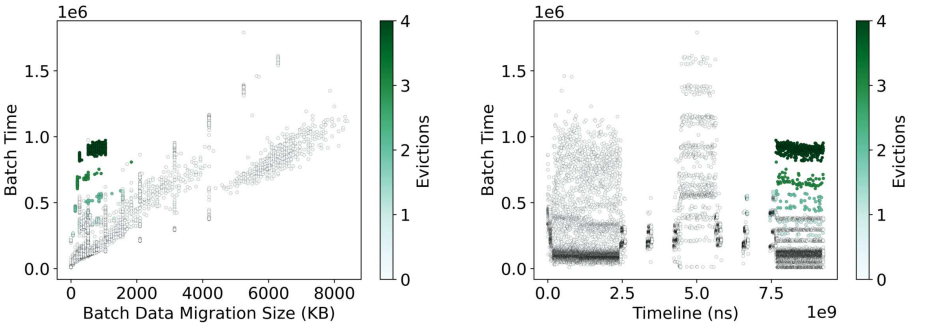
Figure 14: Batch profiles of sgemm with prefetching enabled. The mid-range cost batches are significantly reduced, and the high-end outliers correspond to negative performance impacts from creating and storing DMA mappings.

the high overhead may be caused by the growing of the underlying radix tree, but further investigation is required.

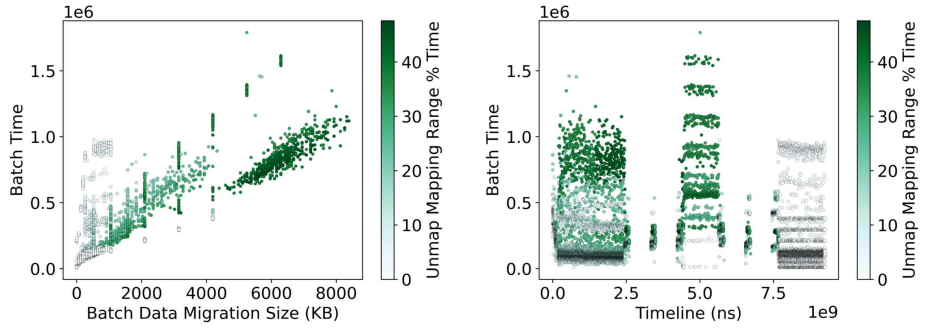
Overall, we confirm our intuition about *when* these batch features may occur and confirm that many of the cost relationships discussed earlier still account for a large quantity of runtime even



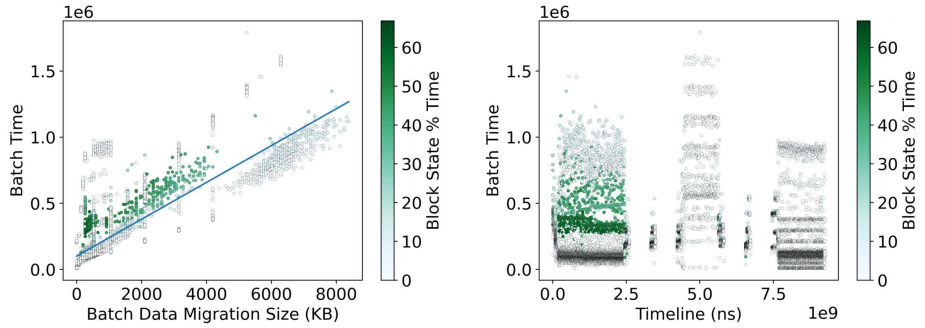
(a)



(b)



(c)



(d)

Figure 15: Batch profiles of `sgemm` by data migration (left) and as a time series (right). Prefetching occurs throughout the execution. Evictions typically occur later in computation. Unmapping and GPU state setup occur regularly throughout the application, and GPU state setup does not always have excessively high overhead.

with eviction and prefetching enabled. Additionally, we find that eviction costs are largely independent of the host OS performance problems and need to be optimized independently. Prefetching can significantly diminish the total number of batches, but the remaining batches include all remaining OS costs, including DMA mapping and host page unmapping.

5.4 Case Studies with HPC Workloads

We use HPC workloads as case studies and show their batch profiles and the corresponding fine-grain fault behaviors. For these experiments, prefetching is enabled and GPU memory is oversubscribed (< 125% GPU memory capacity). Due to space limitations, we only include the Gauss-Seidel and HPGMG benchmarks. We note that while Figure 16c largely shows the order of batch occurrence in time, multiple batches may be condensed in a cluster of points with a much smaller time in Figures 16a and 16b.

Table 4: Batch and Kernel Execution Times

Benchmark	No Prefetch		Prefetch	
	Batch	Kernel	Batch	Kernel
Gauss-Seidel	60.477s	66.393s	15.340s	19.550s
HPGMG	32.384s	40.472s	7.261s	14.879s

Table 4 presents the overall application performance. Aggregate batch times are smaller than kernel times as they exclude the initial interrupt time (negligible) and GPU time spent working with in-memory data. With modest oversubscription, prefetching improves the kernel performance by 3.39x and 2.72x for Gauss-Seidel and HPGMG, respectively. Such performance gain suggests certain amounts of prefetching page hits. In general, the performance gain from prefetching is expected to decrease as the percentage of oversubscription increases and more evictions are involved.

For Gauss-Seidel, the batch time is small at the beginning of execution without intensive prefetching or evictions. We observe larger batch time and increasing number of prefetches around 0.5 second, consistent with the observed larger migration sizes for batches with prefetching. Coincidentally, we observe more evictions begin to occur just before prefetching. This is because eviction creates new opportunities for prefetching to occur - freshly paged-in VABlocks have a high chance of triggering prefetching with subsequent accesses. Respectively, the fine-grain fault behaviors in Figure 16c exhibit contiguous batches allocating and evicting pages in similar, large page ranges. This indirect relationship between allocation, eviction, and prefetching can be observed during the rest of the workload execution.

For HPGMG, there are few faults during the setup phase so the x-axis is cut in Figures 17a and 17b to make drawing space for later execution. We observe similar coincidence between intensive prefetches and increasing evictions in about four segments in Figure 17. We observe the same relationship between allocation, eviction, and prefetching that was present in Gauss-Seidel. Another interesting observation is that Figure 17c clearly manifests the Least-Recently-Used (LRU) replacement policy for page eviction. In practice, LRU policy is essentially "earliest allocated pages" for these sufficiently dense access because the UVM driver has no information about page hits. The first large number of evictions target the first allocated pages, illustrated by the green vertical rectangle at the beginning of the execution. The later evictions similarly evict the remaining earliest allocated pages. This LRU policy may not be optimal, as some evicted pages are needed shortly and must again be migrated back to GPU.

6 DISCUSSION AND CONCLUSION

This work examines how faults are generated by NVIDIA devices and handled by the UVM driver. We identify the key cost components with unexpected performance characteristics in the UVM fault path. We examine 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 investigation into the systems software performance concerns for UVM and a proxy study for HMM and future interfacing vendors/devices. Below we summarize the key findings, discuss them in a wider scope, and discuss potential future work.

Key Driver Costs. *Data movement* contributes only a small amount of overall cost in contrast to expectation. This suggests that improvements to basic hardware, such as interconnect bandwidth and latency, would still improve performance but would not resolve the underlying issues.

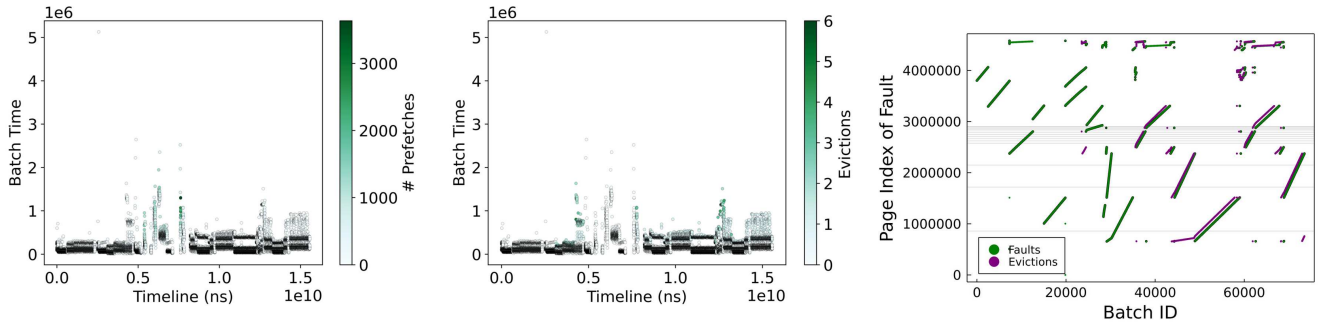
Duplicate faults are an important performance issue that are appropriately managed through limited batch sizes in UVM. Minimizing duplicates is a secondary objective, however. The primary objective is to accept as many unique faults as possible to reduce the total number of batches. A simple improvement could be to tune batch size based on the number of duplicate faults received.

Host OS operations, particularly unmapping CPU pages on the fault path, contribute significant overhead. Some user code parallelization schemes can exacerbate these costs.

CPU Unmapping and DMA Setup are particularly important costs, as they take place on the fault path and are handled by the host OS in HMM and UVM. In the case of HMM, the cost incurs on all implementing devices/vendors. Likely, page-unmapping was never intended to happen in frequent bursts with real-time constraints as is the case with UVM and HMM. As HMM is common code, and UVM is commonplace today, further investigation is necessary to determine if this functionality can be improved to (1) incur less overall overhead and (2) avoid excessive costs based on the chosen parallelization of user applications. Alternatively, performing these operations asynchronously and preemptively may be preferable when an application shifts to GPU compute.

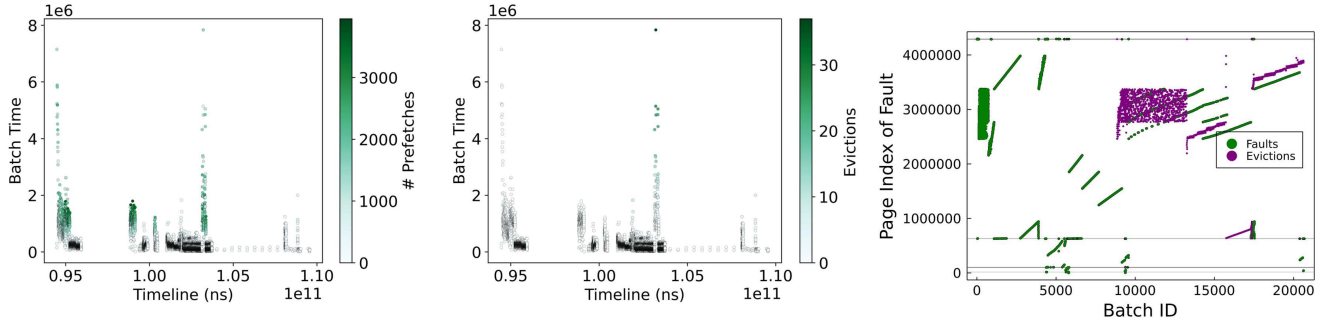
Driver Serialization. Code generation and device-level throttling limit the generation of faults from each SM and ensure batches representing every SM. Consequently, the GPU is generally stalled during driver fault processing, leading to highly synchronous behavior between the CPU and GPU with little overlap and high latency cost. 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 straightforward 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. While these workload features are specific to NVIDIA GPUs, any vendor implementing HMM for parallel devices will encounter similar concerns and delays.



(a) Prefetches in Timeline (b) Evictions in Timeline (c) Faults vs Batches

Figure 16: Batch profiles and fault behavior of Gauss-Seidel with about 16% oversubscription. (a): batch profiles with prefetching. (b): batch profiles with eviction. (c): fault behavior. For simplicity, (c) dismisses the prefetching information and shows batch ID instead of time.



(a) Prefetches in Timeline (b) Evictions in Timeline (c) Fault vs. Batches

Figure 17: Batch profiles and fault behavior of HPGMG with about 25% oversubscription.

Prefetching and Eviction. Prefetching and eviction are UVM-specific features that improve performance and provide additional programmer flexibility, respectively. Eviction creates levels of performance per VABlock evicted. More cost-effective oversubscription requires optimization independent of host OS problems as the underlying performance issues stem from algorithmic issues and user applications, not host OS interference. However, the combined OS and eviction costs are exceedingly high.

Prefetching is effective because it eliminates large numbers of batches and their associated overhead. However, prefetching cannot mitigate batches with high DMA and CPU unmapping overhead, increasing the impact of these costs in real workloads. Because prefetching is constrained to within VABlock, it cannot eliminate or preempt these high-cost batches. Methods, such as increasing the prefetching scope to more than one allocation and asynchronous prefetching, could mitigate these issues but may also complicate eviction. These two features must be codeveloped for devices that implement both.

Applicability to Other UVM-like Implementations. In general, findings presented in this work should reflect similarly designed hardware/software systems, particularly for other device drivers that will serve as backends for HMM. First, we expect other systems would take a batching approach as this is an effective optimization, making our findings regarding batches, duplicates, and batch sizes generally applicable. These design decisions are critical to the overall performance, as the system software overhead, instead of the hardware data transfer time, is the dominant cost. Second, any functionality invoking the Linux kernel is prone to generating high software overhead because the kernel is not designed to process complex operations such as random page unmapping for VABlocks with real-time performance. Finally, our findings regarding fault origin distribution indicate that there is room for system architects to explore driver parallelism and load balancing complying with the VABlock-based execution order. With appropriate load-balancing in fault servicing, parallelism could potentially hide the latency of some system-side operations and allow faster bulk fault servicing.

ACKNOWLEDGEMENTS

This work is supported in part by the U.S. National Science Foundation under Grants CCF-1551511 and CNS-1551262. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] [n.d.]. *High Performance Geometric Multigrid.* Retrieved July 13, 2021 from "https://crd.lbl.gov/departments/computer-science/par/research/hpgmg/"

[2] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 141–150. https://doi.org/10.1109/IPDPS49936.2021.00023

[3] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 136–150. https://doi.org/10.1145/3123939.3123975

[4] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 596–609. https://doi.org/10.1109/HPCA47549.2020.00055

[5] Natalie Beams, Ahmad Abdelfattah, Stan Tomov, Jack Dongarra, Tzanio Kolev, and Yohann Dudouit. 2020. High-Order Finite Element Method using Standard and Device-Level Batch GEMM on GPUs. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 53–60. https://doi.org/10.1109/ScalA51936.2020.00012

[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. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

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

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

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *ArXiv* abs/1410.0759 (2014).

[10] Steven Chien, Ivy Peng, and Stefano Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (Nov 2019). https://doi.org/10.1109/mchpc49590.2019.00014

[11] Linux Kernel Development Community. [n.d.]. *Heterogeneous Memory Management (HMM).* Retrieved May 25, 2021 from https://www.kernel.org/doc/html/latest/vm/hmm.html

[12] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *High Performance Computing*, Michela Taufer, Bernd Mohr, and Julian M. Kunkel (Eds.). Springer International Publishing, Cham, 489–507.

[13] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 3–10. https://doi.org/10.1177/1094342015593158 arXiv:https://doi.org/10.1177/1094342015593158

[14] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay Between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/3307650.3322224

[15] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2020. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. https://doi.org/10.1109/ipdps47924.2020.00054

[16] R. Gayatri, K. Gott, and J. Deslippe. 2019. Comparing Managed Memory and ATS with and without Prefetching on NVIDIA Volta GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 41–46.

[17] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing Large Graphs on GPUs with Unified Memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1119–1133. https://doi.org/10.14778/3384345.3384358

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

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

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

[21] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. https://doi.org/10.1145/3373376.3378529

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

[23] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt. 2014. An investigation of Unified Memory Access performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

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

[25] K. V. Manian, A. A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, and D. K. Panda. 2019. Characterizing CUDA Unified Memory (UM)-Aware MPI Designs on Modern GPU Architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs* (Providence, RI, USA) *(GPGPU '19)*. ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/3300053.3319419

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

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

[28] J. M. Nadal-Serrano and M. Lopez-Vallejo. 2016. A Performance Study of CUDA UVM versus Manual Optimizations in a Real-World Setup: Application to a Monte Carlo Wave-Particle Event-Based Interaction Model. *IEEE Transactions on Parallel and Distributed Systems* 27, 6 (2016), 1579–1588.

[29] NVIDIA. [n.d.]. *Open GPU Documentation.* Retrieved May 25, 2021 from https://nvidia.github.io/open-gpu-doc/

[30] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (1995), 1–19. https://doi.org/10.1006/jcph.1995.1039 http://lammps.sandia.gov.

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

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

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

[34] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. 2010. Speeding up Nek5000 with Autotuning and Specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing* (Tsukuba, Ibaraki, Japan) *(ICS '10)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/1810085.1810120

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

[36] Q. Yu, B. Childers, L. Huang, C. Qian, H. Guo, and Z. Wang. 2020. Coordinated Page Prefetch and Eviction for Memory Oversubscription Management in GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 472–482. https://doi.org/10.1109/IPDPS47924.2020.00056

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

[38] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang. 2020. HPE: Hierarchical Page Eviction Policy for Unified Memory in GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2461–2474. https://doi.org/10.1109/TCAD.2019.2944790

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

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

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 5.9.16-200.fc33.x86_64. The system has an AMD Epyc 7551P 32-Core CPU with 128GB of memory.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪ https://zenodo.org/badge/latestdoi/356388244
Artifact name: Instrumented Driver, Experiments, and
↪ Evaluation Tool
Citation of artifact: Tyler Allen. (2021).
↪ tallendev/uvm-eval: SC2021-Artifact (v0.1).
↪ Zenodo. https://doi.org/10.5281/zenodo.5148930
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Titan V GPU, MD Epyc 7551P 32-Core CPU, 128GB DDR4

Operating systems and versions: Fedora 33 running 5.9.16-200.fc33.x86_64, CUDA 11.2, and NVIDIA Driver version 460.27.04

Compilers and versions: GCC 10.2.1 and NVCC cuda_11.2.r11.2/compiler.29373293_0

Applications and versions: HPGMG-FV 0.3, UVM-modified CUDA BabelStream, SGEMM-CUBLAS, and Several Synthetic Kernels

Libraries and versions: CUBLAS 11.2

URL to output from scripts that gathers execution environment information.

```
https://github.com/tallendev/uvm-eval/blob/master/Au
↪ thorKit.txt
```