Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis

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

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

***Abstract*—**
With GPUs becoming ubiquitous in HPC systems, NVIDIA's Unified Virtual Memory (UVM) is being adopted as a measure to simplify porting of complex codes to GPU platforms by allowing demand paging between host and device memory without programmer specification. Much like its storage-based counterparts, UVM provides a great deal of added usability at the cost of performance due to the abstraction and fault-handling mechanisms. This is preventing HPC systems from being used efficiently and effectively and decreases the overall value of GPU-based systems.

To mitigate the cost of page fault stall time, NVIDIA has introduced a prefetching mechanism to their UVM system. This prefetcher infers data ahead-of-time based on prior page fault history, hoping to satisfy faults before they occur. Such a prefetcher must be cleverly designed and efficient, as it operates under the constraints of a realtime system for providing effective service. Additionally, the workload is quite complex due to the parallel nature of GPU faults, as well as page fault serialization and fault source erasure within the driver. The current prefetching mechanism uses a density-prefetching algorithm to offset the side-effects of receiving page faults in parallel. While this prefetching can be very effective, it also has a negative impact on the performance of GPU oversubscription.

In this paper, we provide a deep analysis of the overhead caused by UVM and the primary sources of this overhead. Additionally, we analyze the impact of NVIDIA's prefetching and oversubscription in practice on different workloads, and correlate the performance to the driver implementation and prefetching mechanism. We provide design insights and improvement suggestions for hardware and middleware that would provide new avenues for performance gain.

***Index Terms*—GPU, demand paging, UVM, performance**

I. Introduction

Graphics Processing Units (GPUs) are now widely employed in supercomputers and data centers and provide the majority, of the total computational power to accelerate traditional scientific and emerging machine learning workloads. The recent development of Unified Virtual Memory (UVM) technology for NVIDIA GPUs further enhances programmer productivity by providing a single memory space, and automating memory management and data migration between CPU host and GPU device physical memory modules. In addition, it supports oversubscription of GPU memory through demand paging, emulating CPU-sized memory spaces with significantly smaller GPU memory. With UVM, programmers are freed from tedious and error-prone manual memory management and data transfers instructions. The high productivity of GPU computing enables domain researchers to leverage GPU computational power to advance and discoveries at a faster speed for broader ranges of scientific applications.

However, UVM introduces significant overhead. Similar to traditional virtual memory, UVM maps virtual pages to physical pages. Linux-like page table mappings are kept on the GPU for general compute. When a page on demand encounters a miss on the GPU page table walk, a far-fault occurs, which raises an interrupt to the UVM driver and the host for handling. Far-fault handling is costly as it involves multiple round trips between the host and device for page table updates, data migration, and software processing. Far-faults are a significant performance bottleneck for situations where execution warps are stalled to wait for the data. Prior work indicates that the cost of a far-fault is 30-45 µs [1], and significantly higher under oversubscription due to the extra cost of page evictions.

Currently, the costs of UVM can be mitigated but still has substantial overhead. NVIDIA has introduced runtime prefetching as the primary way to hide latency within UVM. In figure 1, we use page-touch kernels to make four key observations about UVM and prefetching: (1) cumulative data access latency without prefetching generally increases one or more orders of magnitude with UVM in comparison to explicit direct management by programmers, (2) when all data fits in GPU global memory, prefetching reduces the cost significantly, but the overall time can still be several times higher than the baseline, (3) once the GPU global memory is oversubscribed, data access latency dramatically increases by another order of magnitude depending on access pattern, and (4) prefetching can aggravate the performance issues after oversubscription. Ideally, prefetching can eliminate all the UVM overhead regardless of memory requirements. This large performance gap motivates us to locate the root cause of performance gap, and identify strategies for improved prefetching and latency hiding.

Researchers have begun to address the overhead of UVM access. A few empirical studies [2], [3] use experimental data to quantitatively evaluate the performance of various access patterns with the default prefetching and eviction algorithms, as well as improved batching, eviction, and prefetching so-

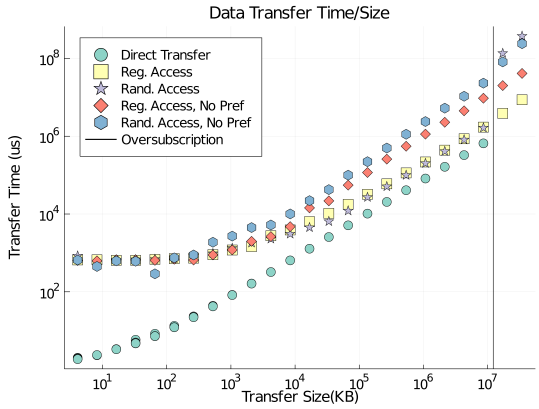


Fig. 1: UVM access incurs one or more orders of magnitude greater latency than direct data transfer, when comparing explicit data transfer to page-touch kernels.

lutions [1], [4]–[7]. This prior work reveals the prefetching and eviction algorithms cause performance issues, but lacks in-depth insight for locating the root causes and guiding the design of effective latency-hiding prefetching and eviction.

Our work seeks to uncover the deeper costs and behaviors of UVM components, to offer a quantitative explanation for performance characteristics at a lower level than seen in prior work. The main contributions of this work are:

- We provide an in-depth analysis of the cost of demand paging.
- We examine application access patterns from the perspective of the driver to gain insight into the workload required for eviction and prefetching.
- We examine the effectiveness of the current prefetcher in terms of runtime performance and fault elimination.
- We look into the impact of oversubscription and the root causes for performance degradation.
- We provide suggestions for paths forward on developing effective prefetching and eviction approaches.

II. Related Work

Several works have analyzed the performance of UVM from different perspectives at a high level, but none that focus on the detailed breakdown of the existing implementation and hardware/software interaction that we provide. Several works offer empirical comparison of overall application performance using UVM to their respective direct-transfer implementations with and without oversubscription [3], [8]–[11]. Additionally, there are works that examine the performance of UVM using performance hints and configurations provided by the CUDA runtime [12], or leverage these hints for application-specific optimization [13]. There are also performance comparisons of UVM between the x86 and Power9 architectures, focusing on the difference between NVLINK and PCIe, as well as the ATS hardware provided on the Power9 system [14]. Gu et. al published a series of benchmarks primarily derived from the Rodinia benchmark suite for performing these kinds of evaluations [2].

Prefetching and eviction with UVM are both areas of interest with UVM in prior work. Several works focus on introducing new methods for prefetching, eviction, or both [1], [4]–[7]. The works presented here depend on the assumption that UVM uses some method of hardware-implemented prefetching, and infer mechanics based on empirical results. For testing new methodologies, they implement their new method in simulators. Our work utilizes the existing NVIDIA UVM driver with functional analysis and understanding from the source code, documentation, and corresponding Open GPU documentation [15] provided by NVIDIA. Our performance results are provided by driver instrumentation or NVIDIA-provided profilers.

There are also efforts into fundamental improvements or alternatives to UVM. Kim et. al propose minor hardware and runtime modifications to improve the efficiency of the existing UVM-like system [16]. Griffin offers a hardware/software improvement to page tracking for multi-GPU systems, improving the system's ability to ensure pages are local where they are most needed [17]. Mojumder et al. offers an alternative to UVM, modifying hardware between the host and device so that all memory between host-device acts as shared memory [18]. These works have more in common with DMA and remote mapping, which we do not focus on in our work.

III. Cost of Demand Paging

In this section, we overview the behavior of UVM as well as the scope of our investigation before digging into the baseline costs of UVM without advanced features like oversubscription and prefetching.

A. UVM Overview

UVM supports three page access behaviors. **Paged migration** moves data between devices in response to a page fault, maps the page into the faulter's physical space, and unmaps from the previous location. Such a fault is known as a far-fault [19] and the focus of our work as it is the common behavior for Pascal and later architectures. We limit our discussion on the other two behaviors: **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 **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.

UVM on-demand paging is implemented using GPU hardware and CPU software working in tandem. To integrate with the host OS, the UVM driver is provided as a kernel module for the host OS to extend the virtual memory space and map it to GPU global memory utilizing the host memory layout e.g., 4KB pages for x86 architectures. The driver interfaces with GPU hardware and manages related data structures. Figure 2 presents the architecture of paging mechanism. Page tables are maintained on either side and map virtual pages to the

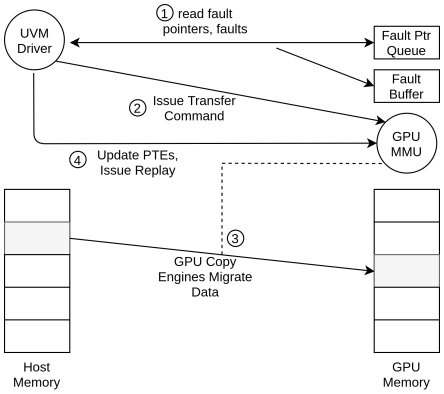


Fig. 2: UVM Fault-Handling Communication Architecture. (1) The UVM driver reads fault pointers out of a queue. The pointers lead to the specific fault entry in the fault buffer. Faults are cached on the host. (2) After processing, the driver will alert the GPU of data to be transferred. (3) The GPU will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

assigned physical space. GPU MMUs, upon a miss on the page table walk, issue an interrupt while the faulted pages in a fault buffer. Upon the receipt of interrupt, the UVM driver is invoked by the OS to access the fault buffer and handle the faults, manage the updates of page tables, and initiate data transfers between the host and device.

Paging at the 4KB granularity creates high management overhead and reduces the utilization of the host-device interconnect. To better utilize the host-device interconnect, amortize the performance impact of transfer latency, and ease management and tracking, UVM adds some additional abstraction on top of the virtual memory space. Specifically, 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 an application. 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.

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 locally. 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 default batch size is 256 faults. Per batch, the driver groups page faults based on VABlocks and *service* the faults. Fault servicing involves memory allocation, updating page tables, data transfer, and possibly issuing one or more fault replays or other operations,

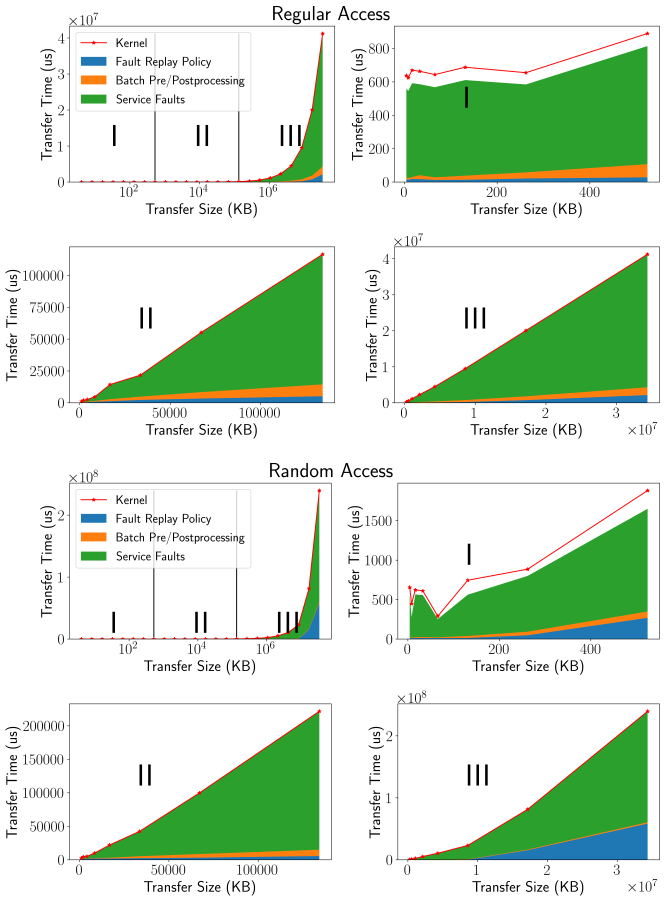


Fig. 3: Fault cost scaling and breakdown at different magnitudes of scale for two access patterns on the same data size. Random tends to be slower, and has shifted proportions.

subject to the *fault replay policy*.

B. Experimental Environment

All experiments are performed on a Titan V GPU with 12GB HBM2 memory using CUDA 11.0 and NVIDIA Driver version 450.51.05. The system has an AMD Epyc 7551P 32-Core CPU with 128GB of memory. In particular, all experiments fit in-core on the host, but oversubscription experiments exceed the GPU memory capacity by some percentage.

Benchmarks used for evaluation include two synthetic benchmarks including a regular and random access pattern, cuBLAS SGEMM [20], Stream (triad-only) [21], Tealeaf [22], HPGMG [23], forward and inverse cuFFT [24], and a cuSparse kernel that converts a dense matrix to a sparse matrix and preforms a sparse matrix multiplication [25]. All applications were developed by the authors unless attributed, in which case the authors ported thee applications to use UVM, with the exception of HPGMG.

C. Cost Overview

To understand the costs associated with demand paging, we 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. We instrument the open-source UVM driver provided by NVIDIA to time the involved operations. Figure 3 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.

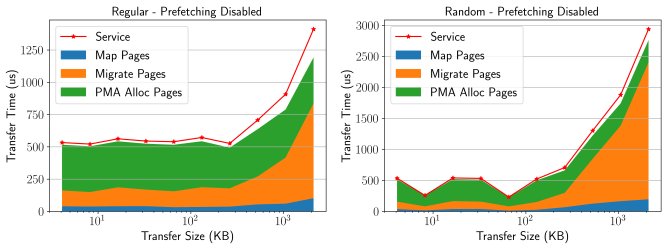
Pre/post processing is shown to be negligible in cost, but functionally important for the fault servicing and replay implementation. Pre-processing first gathers faults from the device, performs basic bookkeeping and logical checks, and sorts them into the appropriate VABlock bins. NVIDIA documentation indicates that the driver uses a circular device-side queue to store a fault *pointer* when a fault occurs [15]. 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 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.

Understanding the total cost calls for deep analysis of the *service* and *replay policy* categories. In the following subsections, we delve into the internal design of UVM and analyze these two constituent costs.

D. Service Cost Breakdown

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 4 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



(a) (b)

Fig. 4: 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.

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.

We draw several important insights from these analyses. First, the number of VABlocks in a batch has a great impact on service time. For the same number of page faults, a batch containing fewer fully faulted VABlocks takes much less time than a batch containing VABlocks each with one page fault as operations can be coalesced and performed in bulk. The former has better data locality, better support for coalescing data transfers, and requires less overall allocation/staging operations. Applications with random accesses would incur the most cost by this rationale. Pre-processing enables these optimizations by binning page faults into their respective VABlocks. Second, the batch size affects the cost and the optimal size depends on application access patterns and data requirement. Larger batches have a better chance to have more page faults in the same VABlock, which better utilizes the bandwidth and amortizes migration cost, at the cost of potentially delaying

SMs and accumulating more faults in the fault buffer. The appropriate tuning of batch size may differ on a per-application basis, and would be an interesting area of future study.

E. Replay Policy Cost

Once the UVM driver has serviced all page faults in the current batch and the data are ready on the destination location on GPU, it notifies the GPU to replay far-faults. *Replayable faults* do not block the faulting GPU compute unit, which can continue running non-faulting warps until a `replay` command is received [19]. Thus replayable faults improve latency hiding on GPUs. The replay notification indicates that the original memory access should be tried again. Note that a single fault may need to be replayed multiple times due to hardware fault capacity limitations or software policy.

Deciding when to notify fault replay has some considerations due to trade-offs between latency and replay overhead. It is not necessary for all outstanding faults to be serviced prior to issuing a replay notification, but a notification allows faulting SMs to resume sooner at the cost of additional instances of replay overhead. Furthermore, issuing replays with outstanding faults causes unsatisfied requests to fault again, generating duplicate faults in the fault buffer and more processing for the UVM driver subject to policy. On the other hand, waiting too long to issue replays causes warps to be stalled for a long period of time, which has a negative cascading effect latency hiding.

To balance the tradeoff, the current NVIDIA driver supports four policies. They differ by the condition for issuing a replay notification.

Block policy — all faults for a block 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 — each fault batch has been serviced. This policy has fewer replays with the potential for larger latency for fault resolution.

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 — all faults in the buffer have been serviced. This policy is the extreme case with simple design and long latency.

To demonstrate the impact of the chosen batching policy, we ran trials with the "Batch Policy." The primary difference between this policy and the default is that the fault buffer is no longer emptied after each batch, meaning that the policy cost now only accounts for the act of issuing a replay. As expected, figure 5 shows the tradeoff when compared to 3. We omit the random access pattern here, but note that it behaves similarly with roughly twice the service cost. The increased

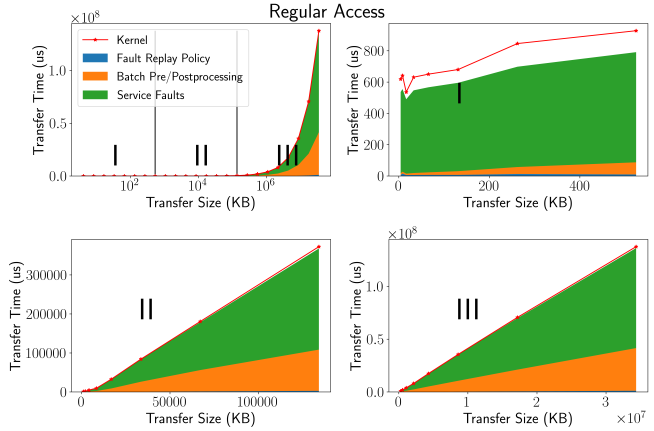


Fig. 5: This figure demonstrates the same experiment as 3, except with the Batch Policy as the replay policy. Note that the replay policy cost is severely diminished, while the preprocessing cost is greatly increased.

preprocessing cost indicates that flushing the buffer has the intended latency-reducing effect.

IV. Prefetching Challenges and Page-Level GPU Access Patterns

A. Prefetching Design Constraints and the Tree-Based Algorithm

To hide the latency of demand fetch of pages, the UVM driver utilizes a prefetcher to migrate pages from the host to device before they are used. UVM faces the fundamental prefetching challenge of a finite lead time, the requirement to effectively hide latency. The implementation of a prefetcher in *software* adds several unique challenges. These challenges limit the prediction algorithms and migration strategies, which consequently decide the prefetching performance.

First, the lead time is more stringent than hardware prefetching due to the greater cost for executing in software, larger data size, and different interconnect characteristics. To resolve the conflict between small lead time and the costly delays, the UVM prefetcher should try to prefetch a large volume of data where possible to better utilize the H-D interconnect bandwidth, reduce the overall number of faults, and amortize the prefetching cost.

Second, the UVM page prefetcher only has coarse-grain, partial information of page accesses. Particularly, the only information specific is the address that originated the fault. The driver lacks sufficient information for correlating faults with their generating GPU core/thread. Poor prediction from limited information may result in fetching a large amount of unwanted data, wasting H-D bandwidth. Consequently, UVM prefetching is left with the best effort option to support common applications.

Third, UVM page prefetching adds transfer overhead that could be wasted if prediction is poor. Unlike CPU main memory, GPU global memory is several orders of magnitude smaller, i.e., 10s of GB vs 100-1000GB. When GPU memory

is oversubscribed, aggressive prefetching wastes bandwidth for both prefetching and later eviction.

To address these challenges, the UVM prefetcher for NVIDIA GPUs presently adopts a simple two-stage prefetching mechanism that is invoked per-VABlock with at least one faulted page within a batch, as described in section III. First, 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. (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" [26] as it is labeled in the prefetcher source code, or sometimes referred to as a tree-based prefetcher in other works [1]. 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.

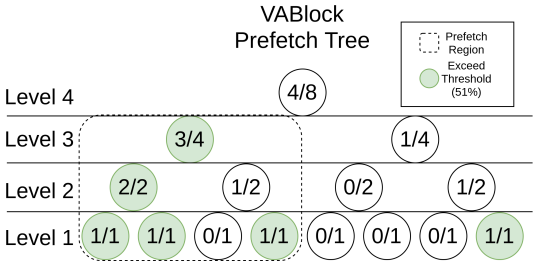


Fig. 6: Illustration of the current UVM page prefetching tree concept. The access density threshold is 51%, the default value. Leaves represent 4KB pages in a 2MB address block. At each level, values of the child node are accumulated in the parent. The prefetch region is the largest such region that 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 6 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.

B. Complex GPU Access Patterns

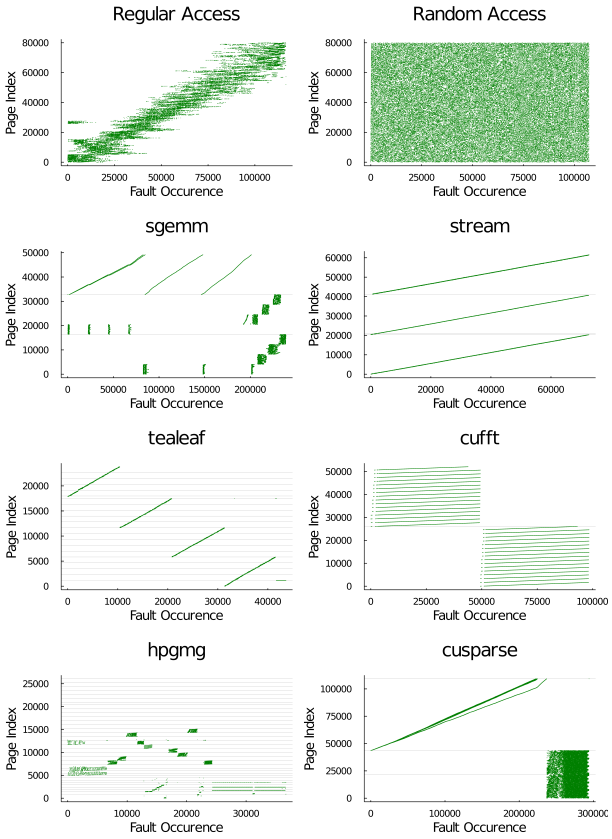


Fig. 7: Application access patterns with prefetching disabled. The page index is the virtual memory page corresponding to the fault address, adjusted so that there are no gaps in the virtual memory space. Fault occurrence is the relative order that pages were processed by the driver.

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 7 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 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

TABLE I: 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.

ordering due to the nondeterminism 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.

C. Effectiveness of Tree-Based Algorithm

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.

We analyze the impact of access patterns by examining our series of benchmarks. Referring to table I, 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 suboptimal prefetching and driver overhead still harm the overall performance. In terms of the number of faults eliminated, **fault coverage**, 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.

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.

V. The Game-Changer: Oversubscription

Overstepping the GPU memory limitations into oversubscription adds complexity and performance concerns due to eviction and its interactions with memory allocation, prefetching, and application access pattern. It is helpful to conceptualize the GPU memory in this scenario as a fully-associative cache for CPU memory where data must be evicted once the capacity of the cache is exceeded. The cache-line size can be treated as a VABlock, as the VABlock is the size that memory is allocated on the GPU, although the full cache-line is not migrated simultaneously. Once oversubscription starts, we have to start considering cache evictions. Because the UVM system is imposing a remote software cache abstraction over large chunks of GPU memory, there is plenty of room for performance concerns.

A. The Cost of Eviction

1) Eviction in UVM: NVIDIA implements an 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.

The UVM driver uses least-recently-used eviction. 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.

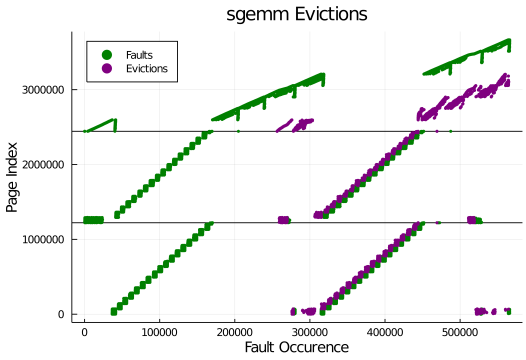


Fig. 8: This figure demonstrates sgemm with a problem size that requires approximately 120% of GPU memory. Notably, we show evictions at the relative time step they are issued. Evict and re-fault is a worst-case performance scenario.

2) Direct and Indirect Costs:

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.

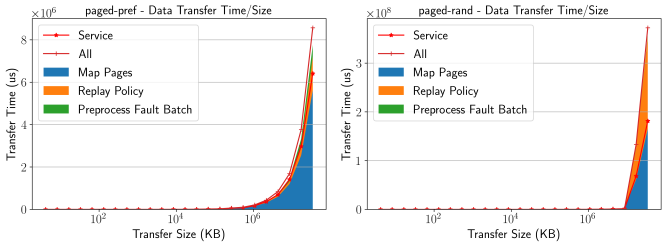
Indirect Costs In terms of specific cost changes indirectly created by eviction, we observe the following:

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 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 8, 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.

Prefetching can fetch data that will not be used prior to eviction. For example, we will move a minimum of 64KB due to the page upgrade size. With prefetching disabled, performance improves after prefetching is disabled, demonstrating the cost of prefetching to effective oversubscription. This can be seen by comparing figure 9 to figure 3 from the prior section.

Delays due to frequent eviction cause larger backlogs of faults in the fault buffer, requiring increased time to flush the fault buffer before replays and accumulating the time spent handling the fault replay policy referred to in section III.



(a) (b)

Fig. 9: 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.

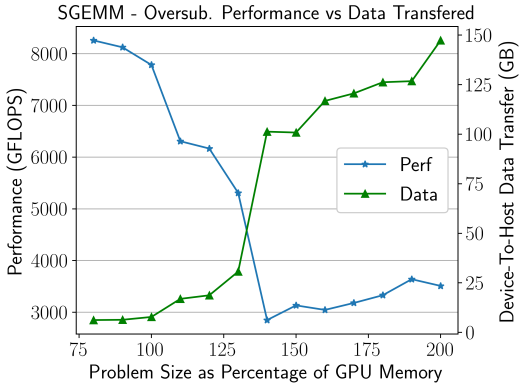


Fig. 10: This figure shows the parallel increase in data requirement as compared to compute rate for the `sgemm` kernel.

3) 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 9, 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

TABLE II: 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 10, performance degrades as the number of pages evicted per fault increases.

as well as the access pattern itself. Since the GPU behaves as a cache, it follows similar locality principles as a traditional hardware cache. This is demonstrated in figure 10, 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 8 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 II, where we can see that the number of pages that require migration due to eviction rises as a partial indicator of performance.

VI. Discussion and Conclusions

A. Challenges in Effective Prefetching and Eviction

Prefetching and eviction have related costs 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. 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 review some challenges present in the current architecture that make these issues difficult to solve.

Prefetching. The primary cost of prefetching is in additional data migration. While the data that is migrated already has physical memory allocations associated with it, the cost of migrating that data increases the latency of data movement and slows the handling of further faults.

Density prefetching has limitations when eviction is involved, because there is no guarantee that any prefetched data will actually be used. While a key advantage of density prefetching is its ignorance of precise fault order, it loses a lot of information about spatial locality. Large prefetching regions can be triggered by a handful of faults separated by large temporal spans, demonstrating no real spatial locality but fetching the data anyway. Traditional hardware techniques are difficult to use in this situation because they rely on precise fault orderings requiring per-core access information not available to the UVM driver. Likewise, it is difficult for application developers to tune their application against the prefetcher due to the complexity of timings and data dependencies.

Eviction. Eviction is a very difficult problem because it has its own algorithmic component, as well as a dependency on the memory allocation functionality. Algorithmically, the implementation is still dependent on page fault information, which is insufficient. The granularity of evictions also impacts its performance, which is locked in UVM at the VABlock level.

The LRU algorithm used for eviction has some internal performance contradictions. As discussed in prior sections, the eviction mechanism relies on page faults to a specific VABlock to upgrade that VABlock in the LRU list. This has two key implications: (1) Data that is accessed on the GPU but does *not* cause a page fault because the page is present will not upgrade its location in the LRU list. (2) VABlocks that are fully resident on the GPU will never be upgraded in the LRU list until they are evicted and re-faulted. This has negative implications for hot data at both the page and VABlock granularity. The hottest data will theoretically be migrated to the GPU the fastest, after which it will descend to the bottom of the list towards eventual eviction, requiring the data to be re-paged in later.

Addressing allocation granularity, 2MB blocks may be too coarse for allocations and evictions for irregular applications. While 4KB granularity is very small, irregular applications may not even have locality at the 4KB granularity. This allocation size can lead to many evictions and inefficient use of GPU memory.

B. Potential Paths for Better Prefetching and Eviction

There are a few possible paths for prefetching and eviction that could offer improved performance, or at least additional tradeoff opportunities.

Increased fault origin information. The GPU presently provides quite a bit of 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.

Flexible memory allocation granularity. As discussed before, irregular applications may benefit from a tuneable parameter allowing different sized memory allocations. This could allow for greater on-GPU memory utilization and reduce the overall number of evictions, as well as allowing for greater prefetcher understanding of desired migrations. However, this would likely be a difficult change as it potentially requires

more complex μTLB implementations and highly flexible driver implementation.

GPU memory access-aware eviction. NVIDIA has included support for multiple-granularity access counters for GPU-level memory access on GPUs since the Volta architecture [27]. This is an interesting feature that is not currently being utilized but could potentially be used for smarter and more effective eviction. This idea is explored and simulated in Ganguly et al. [4], but has not been explored on a real system. This information opens the door to many types of eviction of different complexity. This information could also potentially be used for better prefetching inference, assuming the additional data access and transfer does not have prohibitive overhead.

Adaptive prefetching. The existing mechanism has demonstrated that it is quite capable depending on the circumstance. Using existing information, the driver could adapt some simple heuristics to adaptively tune prefetching. For allocation sizes under the GPU memory limitations, there is little reason not to use highly aggressive prefetching to emulate the direct transfer. In contrast, oversubscribed sizes could disable prefetching entirely, or infer from the fault/eviction load how effective prefetching is and tune the prefetching threshold accordingly.

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] D. Ganguly, Z. Zhang, J. Yang, and R. 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*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 224–235. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322224

[2] Y. Gu, W. Wu, Y. Li, and L. Chen, "Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus," 2020, arXiv:2007.09822.

[3] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "A quantitative evaluation of unified memory in GPUs," *The Journal of Supercomputing*, vol. 76, no. 4, pp. 2958–2985, nov 2019.

[4] D. Ganguly, Z. Zhang, J. Yang, and R. 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.

[5] 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)*, 2020, pp. 472–482.

[6] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "Hierarchical page eviction policy for unified memory in gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 149–150.

[7] ——, "Hpe: Hierarchical page eviction policy for unified memory in gpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2461–2474, 2020.

[8] K. V. Manian, A. A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, and D. K. Panda, "Characterizing cuda unified memory (um)-aware mpi designs on modern gpu architectures," in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU '19. New York, NY, USA: ACM, 2019, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/3300053.3319419

[9] 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*, vol. 27, no. 6, pp. 1579–1588, 2016.

[10] M. Knap and P. 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*, vol. 75, pp. 7625–7645, Nov. 2019. [Online]. Available: https://doi.org/10.1007/s11227-019-02966-8

[11] 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)*, 2014, pp. 1–6.

[12] S. Chien, I. Peng, and S. Markidis, "Performance evaluation of advanced features in cuda unified memory," *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, Nov 2019. [Online]. Available: http://dx.doi.org/10.1109/MCHPC49590.2019.00014

[13] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W. mei Hwu, "Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus," 2020, arXiv:2006.06890.

[14] 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)*, 2019, pp. 41–46.

[15] NVIDIA, "Open gpu documentation." [Online]. Available: https://nvidia.github.io/open-gpu-doc/

[16] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. 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.

[17] 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)*, 2020, pp. 596–609.

[18] S. A. Mojumder, Y. Sun, L. Delshadtehrani, Y. Ma, T. Baruah, J. L. Abellán, J. Kim, D. Kaeli, and A. Joshi, "Mgpu-tsm: A multi-gpu system with truly shared memory," 2020, arxiv:2008.02300.

[19] 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)*, 2016, pp. 345–357.

[20] NVIDIA. cublas. [Online]. Available: https://developer.nvidia.com/cublas

[21] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds. Cham: Springer International Publishing, 2016, pp. 489–507.

[22] U. M.-A. Consortium. (2016) Tealeaf. [Online]. Available: https://github.com/UK-MAC/TeaLeaf_CUDA

[23] N. Sakharnykh. High-performance geometric multi-grid with gpu acceleration. [Online]. Available: https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/

[24] NVIDIA. cufft. Test. [Online]. Available: https://developer.nvidia.com/cufft

[25] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*, 1st ed., W. Zhang and C. Zhao, Eds. GBR: Wrox Press Ltd., 2014.

[26] 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*, 2008, pp. 22–27.

[27] NVIDIA. Nvidia tesla v100 gpu architecture. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf