UVMMU: Hardware-Offloaded Page Migration for Heterogeneous Computing

Jihun Park*, Donghun Jeong†, and Jungrae Kim‡
*†*SK Hynix* ‡*dept. of Semiconductor Systems Engineering*, *Sungkyunkwan University*
E-mail: *jihun.park1012@gmail.com, †dounghun22@g.skku.edu, ‡dale40@skku.edu

***Abstract*—In a heterogeneous computing system with multiple memories, placing data near its current processing unit and migrating data over time can significantly improve performance. GPU vendors have introduced *Unified Memory (UM)* to automate data migrations between CPU and GPU memories and support memory over-subscription. Although UM improves software programmability, it can incur high costs due to its software-based migration. We propose a novel architecture to offload the migration to hardware and minimize UM overheads. Unified Virtual Memory Management Unit (UVMMU) detects access to remote memories and migrates pages without software intervention. By replacing page faults and software handling with hardware offloading, UVMMU can reduce the page migration latency to a few μs. Our evaluation shows that UVMMU can achieve $1.59\times$ and $2.40\times$ speed-ups over the state-of-the-art UM solutions for no over-subscription and 150% over-subscription, respectively.**

***Index Terms*—GPU, Unified Memory, Page Migration and Fault**

I. Introduction

Heterogeneous computing combines the best among CPUs and accelerators (e.g., GP-GPU) to achieve versatility, high performance, and low energy consumption. Applications partition their jobs and execute on the best fitting processor, which requires data sharing among the processors. Meanwhile, a high-end accelerator typically has a separate memory to meet its high data bandwidth requirement. Fetching data from the local memory provides low latency, high bandwidth, and low energy consumption. Therefore, it is important to migrate data to its current processing unit for high performance and energy efficiency in heterogeneous computing.

Conventionally, data sharing between CPU and GPU is based on user-directed memory copies. A programmer allocates memory regions in both memories and uses explicit commands (e.g., cudaMemcpy()) to copy data back and forth between the regions. However, this simple approach has several issues. First, the programmer is responsible for the tedious and error-prone data migration job. After the human endeavor, the copy operation blocks a kernel execution to serialize and significantly increases the total execution time. As GPU memory comprises fewer DRAM chips, programmers must partition data and tasks to fit into the limited memory capacity.

NVIDIA introduced *Unified Memory (UM)* to improve programmability [1]. With a single virtual address space, CPU and GPU can share data using the same pointer. It also allows over-subscription to GPU memories. GPU applications can use more memory than the physical capacity of GPU memory, using the CPU memory as backing storage.

This work was partly supported by National Research Foundation of Korea(NRF) (No. 2020R1C1C1011419, 40%) and Institute of Information & communications Technology Planning & Evaluation(IITP) (No. 2021-0-00479, 40% / 2021-0-00863, 20%) grant funded by the Korea government(MSIT).

∗† This work was done when the authors were at Sungkyunkwan University.

Although UM improves programmability, it comes with severe performance costs in current implementations. With UM, the role of data migration is now delegated from programmers to the GPU hardware and its driver. When the hardware detects access to a remote page, it raises a fault via a PCIe interrupt and wakes the driver on the CPU. The driver migrates the remote page to the local memory to fix the fault, and replays the instruction. Although it can partially overlap data migration with computation, the software-based fault handling requires a significant latency ($20\mu s \sim 50\mu s$) and stalls GPU execution for many cycles. As a result, executions with UM are significantly slower than ones with user-directed copies. [2]

Industry and academia have proposed various prefetching and eviction mechanisms to reduce UM overheads [2]–[7]. Although they reduce the number of page faults, the significantly long latency of page fault still worsens their execution time than cudaMemcpy()-based executions.

To this end, this paper proposes a novel architecture, called *Unified Virtual Memory Management Unit (UVMMU)*, to minimize the performance costs of UM. Although prior work utilizes hardware to detect remote page accesses and transfer pages between memories, their software-based destination selection significantly increases the overall latency.

UVMMU replaces the software handling with a dedicated hardware IP in determining migration destinations, making page migrations entirely hardware-based. The IP maintains a free frame list to select a destination without software intervention. For memory over-subscription, it selects a victim for eviction and swaps the victim page with the accessed page. This selection is processed entirely by hardware to have a minimal latency, which is further hidden by background candidate generation. As a result, UVMMU can reduce the page migration latency to a few μs.

The main contributions of this paper are as follows:

- We investigate the impact of page fault latency in current fault-based page migration for UM.
- We propose UVMMU, a holistic solution to offload page migration to the hardware. UVMMU eliminates page faults in the migration process and reduces page migration overhead. To the best of our knowledge, UVMMU is the first to propose hardware-only page migration handling in heterogeneous computing systems.
- We improve the performance by $1.59\times$ and $2.40\times$ over the state-of-the-art software-based scheme for no over-subscription and 150% over-subscription, respectively.

II. Background

This section provides background on memory synchronization in heterogeneous computing, using NVIDIA Unified Mem-

978-3-9819263-7-8/DATE23/© 2023 EDAA

ory (UM) as an example.

A. NVIDIA Unified Memory

NVIDIA introduced *Unified Memory (UM)* in CUDA 6.0 to improve programmability [1]. Unified memory is a single address space accessible from any processor in a system. With a special API, cudaMallocManaged(), applications can allocate a managed memory region shared by CPU and GPUs. The processors can collaborate on the same data using a single pointer without explicit data copy. UM alleviates the programmer's burden of managing memory copy operations and improves programmability.

Another strength of UM is memory over-subscription. GPUs have limited physical memory capacity, and programmers often have to partition their data set into the limited capacity. UM utilizes a large virtual address space to accommodate both CPU and GPU memories so that a programmer can allocate beyond the GPU memory capacity using the CPU memory as backing storage [8]. With UM, programmers are relieved from the data management jobs and can improve their productivity.

To support UM, NVIDIA Pascal GPUs added a new field, *Aperture (A)* flag, to their *Page Table Entries (PTEs)* [9]. A page that resides on the local GPU memory has the A flag as zero, whereas a page that resides on a remote memory has a non-zero value. For remote pages, the Physical Frame Number (PFN) in the PTE represents the PFN in the remote memory.

As a GPU core accesses a page for the first time, it results in a TLB miss and creates a request to the GPU Memory Management Unit (GMMU). GMMU receives the request and walks through the page table to find the corresponding PTE. If the PTE indicates a remote memory, the SM raises a fault and freezes the L1 TLB. Any new translation will be stalled until all faults are resolved to keep translations consistent during the page table modification [10]. The CUDA driver running on the CPU receives the page fault via a PCIe interrupt and determines a new location in the local memory. Then *Page Migration Engine (PME)* conducts the page migration. Once the migration is complete, the driver updates the PTEs with zero A flags (indicating local memory) and local PFNs, and the SM replays the fault instruction.

Although UM gives programmers an illusion of a single memory, its implementation incurs high performance costs. Accessing a remote memory raises a fault and requires the driver's software-based handling with significantly large latency. ($20\mu s \sim 45\mu s$ [2]). This long latency cannot be completely hidden by the limited thread-level parallelism inside an SM and leads to frequent SM stalls (Section III). The performance penalty grows even further with memory over-subscription due to page evictions and thrashing.

B. Page Prefetching and Eviction

NVIDIA and academia have proposed various prefetching and eviction mechanisms for UM to reduce the number of page faults and its overheads. Ganguly *et al.* proposed a sequential-local prefetcher that prefetches 64KB chunk of pages where the faulty page resides [3]. They also discovered that the NVIDIA driver utilizes a tree-based neighborhood prefetcher [3]. The tree-based prefetcher partitions a 2MB memory region into 64KB basic blocks and builds a binary tree using the basic blocks as leaf nodes. As demand accesses and prefetching migrate leaf nodes, it prefetches non-leaf nodes with more than 50% leaf nodes are migrated. By dynamically increasing the prefetching granularity, the tree-based prefetcher can deal with different locality granularities.

With memory over-subscription, evicting the right page can be important to reduce page evictions and minimize performance degradation from page-fault handling latency. To prevent frequently accessed pages from eviction, some utilize the LRU (Least-Recently-Used) in their eviction policies. UVMSmart proposed *Sequential-local* policy, which applies LRU at the 64KB basic block granularity. It maintains an LRU page list and evicts the basic block which the LRU page belongs to [3]. Furthermore, they extended the LRU policy to *tree-based* neighborhood eviction. It maintains binary trees among basic blocks within a 2MB memory region and dynamically adjusts the eviction granularity based on the eviction history, similarly to its tree-based prefetching. However, maintaining an LRU list can be too expensive for a hardware implementation [2].

These prefetch and eviction policies can reduce page faults in applications with regular memory access patterns. However, they may worsen the performance of irregular applications. To mitigate this problem, NVIDIA introduced zero-copy to directly access pinned CPU memory without page migration [11]. UVMSmart [4] proposed adaptive page migration that detects cold pages and prevents them from migration using zero-copies. To handle applications with dynamically changing access patterns, UVMSmart went further step to utilize a pattern-detection scheme to select the best policy [5].

C. Related Work

Some studies have optimized GPU-to-GPU data transfer, but they still require software handling. Muthukrishnan et al. [12] proposed GPS, an HW/SW global publish-subscribe model for multi-GPU memory management. Baruah et al. [13] proposed Griffin, a holistic hardware-software solution to migrate pages evenly across the GPUs for load balancing.

There are trials of hardware-based migrations for non-UM systems. Lee et al. [14] proposed hardware-based demand paging for CPU-only systems. Jeong et al. [15] proposed on-demand page copy to reduce host-to-device memory copy overheads. Those works adopted hardware-based migration but did not target UM systems.

III. Motivation

Although the state-of-the-art prefetching and eviction schemes can reduce the number of page faults and performance overheads, executions with UM are still significantly slower than ones with user-directed memory copies. We suspected the worse performance is primarily due to the long latency from software-based fault handling and performed an experiment. We used the modified GPGPU-Sim, a cycle-level GP-GPU simulator, from UVMSmart [16] and run its NVIDIA UM configuration with different fault-handling latencies. More details of the experiment are in Section V-A1. We gradually reduce the

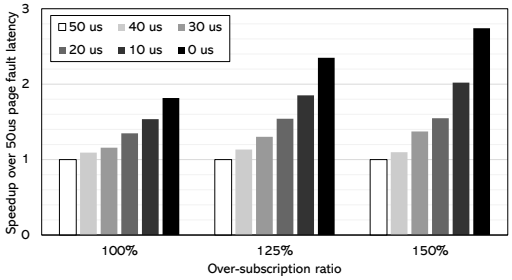


Fig. 1: A comparison of UM execution times with varying page-fault handling latency

latency to zero and see how much performance improvements it provides. Note that prior research reported the lower bound of realistic page fault handling latency is 20μs or 45μs, depending on the processor [2].

Figure 1 shows execution times with different page fault handling latency. The execution times are normalized to ones with 50μs latency under the same over-subscription ratio. Without over-subscription, reducing the latency from 50μs to 20μs speeds up the execution by 35%. Reducing it further to zero, which is not feasible with software, gives an additional 47% speed-up, resulting in a total speed-up of 82%. The performance improvement is more apparent with over-subscription. Decreasing the latency to 20μs provides 54% speed-ups for both 125% and 150% over-subscription. Moreover, a zero latency gives 135% and 174% speed-ups over 50μs latency with 125% and 150% over-subscription, respectively.

The results show that faster page migration can reduce the execution time significantly and imply that prompt page migration using full offloading to hardware can significantly speed up UM executions. Moreover, it implies that the speed-ups will grow further with higher over-subscription ratios. These insights motivated us to develop a novel hardware-based page migration scheme called UVMMU.

IV. UVMMU

We propose a novel architecture called Unified Virtual Memory Management Unit (UVMMU) to minimize page migration overheads of UM. UVMMU replaces the current fault-based page migration, which has a significant latency due to the software involvement. Instead, UVMMU utilizes a hardware IP to migrate pages without software intervention. This section provides an overview of how UVMMU eliminates software intervention in page migration, followed by the details of the software interface and the hardware micro-architecture.

Most UM implementations utilize 64KB as the migration granularity to amortize PCI-e transfer overheads. UVMMU follows the approach and refers to a 64KB memory chunk in the virtual address as a *page-group* and a 64KB memory chunk in the physical address as a *frame-group*.

A. Overview

UVMMU offloads page migration operation from software to hardware entirely. Figure 2 presents the overall flow with the hardware-offloaded page migration. If an SM accesses a remote page, it results in a TLB miss and generates an address translation request to UVMMU (1). The UVMMU

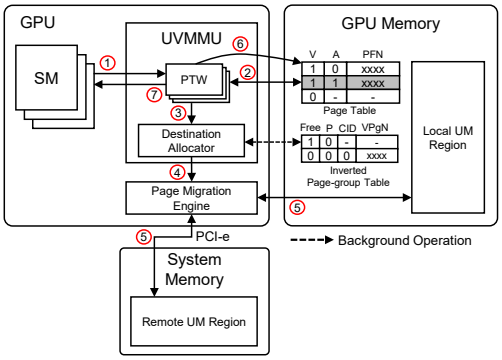


Fig. 2: The overall architecture with UVMMU and the hardware-offloaded page migration flow.

utilizes its *Page Table Walker (PTW)* to find the corresponding PTE in the GPU page table (2). If the A flag in the PTE indicates a remote page, UVMMU initiates a hardware-only page migration instead of returning the PTE to the SM. It first asks an internal module, Destination Allocator (DA), to allocate a memory frame for the migration destination (3). More details of DA are presented in Section IV-D. Then UVMMU requests the PME to perform a page transfer from the remote PFN (from the PTE) to the allocated local PFN (4). If needed, it swaps the victim page with the requested page. PME transfers pages through the PCI-e interconnect (5). Once the transfer is completed, UVMMU updates the PTE with the new PFN and clears the A flag (6). Finally, UVMMU returns the updated PTE to the requesting SM (7). From the SM's point of view, it is a local page with a longer-than-usual page walk time rather than a remote page.

B. Data Structure

UVMMU requires a couple of additional tables to select a migration destination during execution. The software configures the tables before a kernel execution to prepare for the hardware-based migration.

1) Inverted Page-group Table: Inverted Page-group Table (IPgT) stores the mapping from the physical-to-virtual addresses. It is similar to the inverted page table in operating systems, and the only difference is that the granularity is a page-group, not a page. The table is indexed by the physical frame-group number (PFgN), and each entry contains a *free* flag, the mapped virtual page-group number (VPgN), and the GPU context ID (CID). Also, it contains the *protection (P)* flag to prevent migration of non-UM regions (e.g, cudaMalloc()). The table is somewhat large (each entry is 8B and an 80GB GPU memory requires 1.25M entries) and we store it in the GPU memory. UVMMU utilizes this table to find free frame-groups and the virtual addresses of the victim page-groups.

2) Frame-region Available Bitmap: Although UVMMU can find a free frame-group from IPgT, it takes too much time to iterate over the entire table in DRAM. In order to speed up the search, UVMMU introduces Frame-region Availability Bitmap (FrAB), which is a collapsed version of the free flags in IPgT. It groups 32 64KB frame-groups into a 2MB frame-region and collapses the free flags of the frame-groups into the *available* flag of the region. A region has the flag set if it has at least

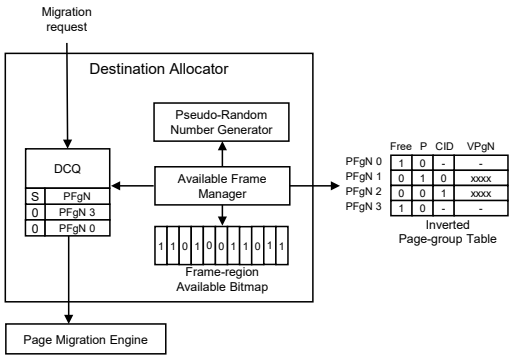


Fig. 3: The micro-architecture of the Destination Allocator and its data structures (Inverted Page-group Table and Frame-region Availability Bitmap)

one free frame-group. If the flag is cleared, the region has no free frame-group, and UVMMU can skip accessing IPgT for the frame-groups. The bitmap is small (40K bits for 80GB physical memory) and stored in an internal SRAM within the GPU.

C. Software Interface

The software is responsible for configuring IPgT and FrAB before kernel executions. On a device bring-up, the CUDA driver allocates IPgT in the GPU memory and puts all available frame-groups to a free list by setting the free flags in IPgT and the available flags in FrAB. The cudaMallocManaged() does not directly change the tables, whereas the cudaMalloc() removes mapped GPU memory frames from the free list by clearing the free flags and setting the protection flags in IPgT. If all frame groups within a 2MB region are mapped, it also clears the available flag in the FrAB.

During a kernel execution, UVMMU hardware selects a free frame-group from the tables and updates them. If there is no available frame-group, the hardware selects a victim and swaps the victim page-group with the requested page-group using the IPgT. When the kernel is complete, the waiting cudaDeviceSynchronize() API writes back modified pages to the CPU memory in preparation for CPU accesses. When the application invokes a cudaFree(), the driver reads the page table to find the mapped frame-groups and adds them back to the free list by updating the IPgT and FrAB.

Because the page management is shared between software and hardware, multiple kernels running in parallel could cause inconsistent tables. For instance, the GPU driver may try to update IPgT for one kernel while the GPU hardware updates the table for another. To avoid inconsistency in such cases, UVMMU can unify the modification paths to hardware. For cudaMalloc(), the software can ask hardware to pop a free frame-group from DCQ via PCIe BAR (Base Address Register) address space. Similarly, on a cudaFree(), the software can ask the hardware to update IPgT and FrAB via the memory-mapped I/O. This single modification path can avoid race conditions and provide atomic updates and consistent page management for multi-kernel scenarios.

D. Hardware Operation

UVMMU is an extension of GMMU, and this section provides the details of its operation, focusing on the *Destination Allocator (DA)* inside UVMMU. Figure 3 illustrates the internal organization of DA. Once a PTW reads a PTE of a remote page (Step ② in Figure 2), it requests DA for a destination frame-group. To save latency, DA generates destination candidates in advance and holds them in a queue called Destination Candidate Queue (DCQ). Each queue entry has a frame-group number and a *swap (S)* flag indicating whether it requires swapping for eviction. As a result, DA can provide a destination in few cycles using the queue. If the destination does not require swapping (i.e., a free frame-group), UVMMU requests PME to fetch the page-group from the remote memory to the local frame-group. To minimize latency, PME fetches the request page (i.e., critical page) first among the pages within the page-group. Once the transfer is complete, UVMMU updates the PTE with the local frame-group number and returns the PTE to the SM.

If the destination requires swapping, UVMMU requests PTE for swapping the page-groups. The remote physical address is from the PTE of the requested page-group, and the local physical address is the destination address from DA. Once swapping is complete, UVMMU updates the PTEs. In addition to updating the PTE of the requested page-group, UVMMU also updates the PTE of the evicted page-group to point the remote PFgN, which was originally occupied by the requested page-group.

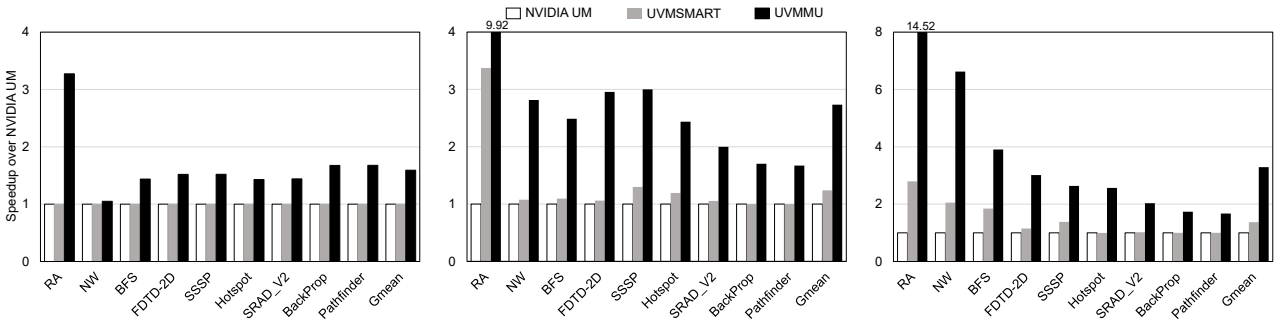
In the background, Available Frame Manager (AFM) in DA starts finding new candidates and filling them into DCQ, if the number of entries in the queue goes below a threshold. In our evaluation, DCQ has 64 entries, and the threshold value is 32 entries. AFM first searches for destination candidates among free frame-groups. It iterates over FrAB to find a region with a free frame-group(s). FrAB utilizes a wide SRAM (e.g., 256-bit) to check multiple regions with single access. If a region has its available flag set, AFM fetches IPgT entries of its frame-groups and add frame-groups whose free and protected flags are 1 and 0, respectively. A 2MB region has 32 64KB frame-groups, thus AFM can fill up to 32 free frame-groups to the queue at a time. AFM repeats this procedure over chunks until DCQ has more entries than the threshold.

If the memory is over-subscribed and AFM cannot find a free frame-group, it moves to find candidates from used frame-groups. It utilizes a pseudo-random number generator to select a random victim from frame-groups. Then it accesses IPgT to check the protection flag whether the frame-group is protected. If not, it adds the victim to the DCQ and sets the swap flag. A random eviction policy may be sub-optimal in some configurations, yet our evaluation shows it provides fair performance with UVMMU (Section V-C1). Meanwhile, its simplicity relieves UVMMU from managing metadata for eviction (e.g., an LRU list).

V. Evaluation

A. Execution Time

1) Environment Setup: We extended the GP-GPU simulator used in UVMSmart [3]–[5]. UVMSmart modified GPGPU-Sim to implement UM [16]. The simulator is configured to represent NVIDIA GTX 1080 Ti (Table I). We modified the simulator to



(a) No over-subscription (b) 125% over-subscription (c) 150% over-subscription

Fig. 4: The speedups of UVMMU and UVMSmart [5] over NVIDIA UM with different memory over-subscription ratios. Note that the Y-axis scale of (c) is twice than others.

replace GMMU with UVMMU and added PCIe models. The migration time is the sum of migration handling latency and PCIe data transfer time. The handling latency is to select a destination page group and configure the PME engine and is 10 cycles with UVMMU and $45\mu s$ with fault-based handling. The PCIe is modeled as delay queues with a $1\mu s$ round-trip time (T_{RTT}) and 16GB/s bandwidth (PCIe gen3 x16).

The baseline architecture is the NVIDIA UM implementation from [3]. We also compared the performance of our architecture with state-of-the-art UM implementation, UVMSmart [5]. The benchmark applications are from UVMSmart [16], which converted 9 workloads from Rodinia [17] and PolyBench [18] into UM. We evaluate execution time for each workload with different memory over-subscription ratios.

2) Results: Figure 4 compares execution times with the different UM implementations. The execution times are normalized to ones with the baseline with the same over-subscription ratio, and the figures show speedups from the baseline.

With no over-subscription (Figure 4a), the baseline and UVMSmart show the same performance. It is because both utilize the same tree-based prefetching, and the scenario has no eviction and does not utilize the smarter eviction policy. On the other hand, UVMMU achieves 59.3% speedup even though it does not prefetch (except for the 64KB migration granularity). NVIDIA UM and UVMSmart can reduce the number of page faults using their tree-based prefetching, yet their large processing delay ($45\mu s$) and freezing L1 TLB generate long stalls in SMs. With faster migration handling process (10 cycles at 1.4GHz) and critical-page-first transfer, page migration with UVMMU takes about $1.3\mu s$ and can hide most of the latency with the WARP-level parallelism.

Figure 4b and 4c show results with memory over-subscription, which migrates more pages due to page evictions. With the smarter prefetching/eviction policies, UVMSmart can reduce the number of page faults more than NVIDIA UM and speed up the execution. It shows 23.9% and 37.2% better performance than the baseline with 125% and 150% over-subscription, respectively. However, both software-based schemes show severe costs from memory over-subscription. UVMSmart with 125% over-subscription, for example, is 75% slower than UVMSmart without over-subscription.

UVMMU shows significantly better speedups with memory over-subscription. With 125% over-subscription, UVMMU shows speedups of $2.73\times$ and $2.21\times$ over NVIDIA UM and UVMSmart with the same over-subscription, respectively. In 150% over-subscription, the speed-ups increase to $3.29\times$ and $2.40\times$. Looking deeper, applications with irregular access patterns (e.g., RA, NW, BFS) show the largest improvements. In the applications, the fault-based schemes cannot accurately prefetch pages due to irregularity, and the long migration latency severely impacts the performance. On the contrary, UVMMU quickly migrates irregularly-accessed page-groups to minimize processor stalls and execution time. For instance, UVMMU improves the execution time of RA $14.52\times$ than the NVIDIA UM in 150% over-subscription. UVMSmart utilizes the zero-copy policy for irregular access patterns and reduces the execution time to some extent ($2.79\times$ of the baseline), yet still $5.2\times$ slower than UVMMU.

TABLE I: The simulated GPU configuration

Core	1481MHz, 28 SMs, 64 warps/SM
Cache Line	128B Line with 4 sectors(32B)
L1 Cache	48KB, 6 Way Associative
L2 Cache	3MB, 16 Way Associative
L1 TLB	128 entries Fully Associative
Memory	11GB 484GB/s GDDR5X
Page Table Walker	64 walkers
PCIe	PCIe 3.0 x16(16GB/s BW, 1us RTT)
Migration Handling Latency	10 cycles (UVMMU), 45 μs (Baseline)

B. Hardware Overheads

Despite the significant performance improvement, UVMMU has small additional overheads. We implemented the logic part of destination allocator in Verilog and synthesized it using Synopsys Design Compiler and Samsung 8nm process. For SRAM-based FrAB, we set its size to 40Kb (corresponding to 80GB physical memory) and used CACTI [19] with ITRS 22nm process (the latest technology supported by CACTI) to estimate the overheads.

The additional area for extending GMMU to UVMMU is estimated to $5800\mu m^2$, which corresponds to 0.0012% of the GTX 1080 Ti die area ($471mm^2$) [20]. It is estimated that UVMMU will only consume 65mW more power than conventional GMMU, which is 0.026% of the 250W TDP of GTX 1080 Ti [20].

C. Sensitivity Studies

1) Eviction Policy: An eviction policy impacts performance by changing the number of page faults. Meanwhile, a sophisticated policy requires maintaining a data structure and complex

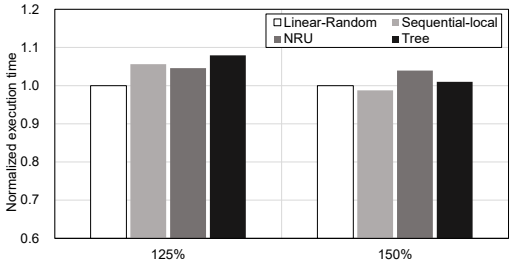


Fig. 5: A comparison of execution times with different eviction policies.

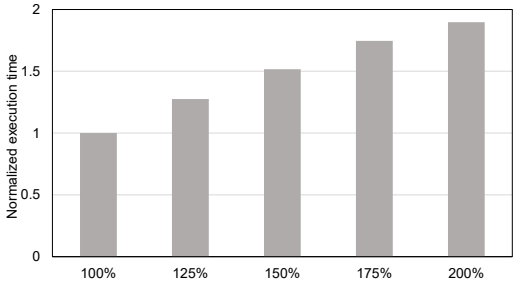


Fig. 6: A comparison of UVMMU execution times with different over-subscription ratios.

hardware. This section evaluates performance with different eviction policies, showing that a simple random policy can provide sufficiently good performance.

UVMMU randomly selects a frame-group using a pseudo-random number generator and evicts the mapped page-group. We call this policy as *Linear-Random* because a page-group is selected randomly, and the pages within the page-group are linearly replaced. We compare *Linear-Random* against *Sequential-local*, *NRU* (Not Recently Used) [21], and Tree-based. *Sequential-local* and Tree-based policies are from UVMSmart [5] and based on the LRU policy (Section II-B).

Figure 5 compares UVMMU performance in geometric mean with the different eviction policies. The Linear-Random policy is not inferior to other sophisticated policies with UVMMU. In fact, it shows the best performance with 125% and 150% over-subscription, except an 1% slow-down than *Sequential-local* with 150% over-subscription. This shows that eviction policy's impacts are significantly less than those reported by fault-based studies. For example, UVMSmart reported that eviction policies affect performance by up to $4.5\times$ with 110% over-subscription. This is because fetching an evicted page again incurs significantly fewer costs with UVMMU.

2) Over-subscription Ratio: A higher over-subscription ratio increases page evictions and can degrade performance significantly. We evaluate the UVMMU performance in geometric mean with different over-subscription ratios (Figure 6). Overall, the UVMMU execution time increases almost linearly with the ratio. For instance, UVMMU shows $1.52\times$ and $1.90\times$ slowdowns to no over-subscription with 150% and 200% over-subscriptions, respectively. In contrast, fault-based solutions have a rapid growth in execution time with increasing ratios (not shown in the figure). UVMSmart, for example, 125% and 150% over-subscription increase the execution time to $1.75\times$ and $2.27\times$, respectively. We failed to run UVMSmart with higher over-subscription ratios(e.g., 175%). We analyze the reason for the rapid growth with UVMSmart is its fault-based migration which cannot efficiently hide the increasing number of page evictions. On the other hand, UVMMU shows a modest growth from reduced page eviction costs, and users may enjoy memory over-subscription with fewer overheads.

VI. Conclusion

Current fault-based UM implementations sacrifice significant performance for improved programmability. The performance costs primarily come from the page-fault handling in software, and most prior works focus on optimizing the software to minimize page faults. We proposed a novel architecture, called UVMMU, to eliminate the faults and offload the entire migration operation to hardware. UVMMU significantly outperforms state-of-the-art UM implementations with minimal hardware overheads with hardware-based rapid migration.

References

[1] M. Harris, "Unified memory in cuda 6." Nov. 18, 2013 [Online] Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.
[2] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *HPCA*, pp. 345–357, 2016.
[3] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory," in *ISCA*, p. 224–235, 2019.
[4] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *IPDPS*, pp. 451–461, 2020.
[5] D. Ganguly, R. Melhem, and J. Yang, "An adaptive framework for oversubscription management in cpu-gpu unified memory," in *DATE*, pp. 1212–1217, 2021.
[6] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "Hpe: Hierarchical page eviction policy for unified memory in gpus," *TCAD*, vol. 39, no. 10, pp. 2461–2474, 2020.
[7] 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 *IPDPS*, pp. 472–482, 2020.
[8] N. Sakharnykh, "Unified memory for cuda beginners." Jun. 19, 2017 [Online] Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.
[9] NVIDIA, "Pascal mmu format changes." Mar. 5, 2016 [Online] Available: https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.
[10] N. Sakharnykh, "Maximizing unified memory performance in cuda." Nov. 19, 2017 [Online] Available: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/.
[11] C. Garg, "Improving gpu memory oversubscription performance," 10 2021.
[12] H. Muthukrishnan, D. Lustig, D. Nellans, and T. Wenisch, "Gps: A global publish-subscribe model for multi-gpu memory management," in *MICRO*, p. 46–58, 2021.
[13] T. B. et al, "Griffin: Hardware-software support for efficient page migration in multi-gpu systems," in *HPCA*, pp. 596–609, 2020.
[14] G. L. et al., "A case for hardware-based demand paging," in *ISCA*, pp. 1103–1116, 2020.
[15] D. Jeong, J. Park, and J. Kim, "Demand memcpy: Overlapping of computation and data transfer for heterogeneous computing," *IEEE Access*, vol. 10, pp. 79925–79938, 2022.
[16] D. Ganguly, "Gpgpu-sim uvm smart." 2018 [Online] Available: https://github.com/DebashisGanguly/gpgpu-sim_UVMSmart.
[17] S. C. et al., "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, pp. 44–54, 2009.
[18] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," 05 2012.
[19] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," vol. 14, jun 2017.
[20] "Nvidia geforce gtx 1080 ti specs," 02 2019.
[21] E. Kabir, N. Akhtar, and S. Mahmud, "An efficient page replacement algorithm," 11 2013.