# Rethinking Page Table Structure for Fast Address Translation in GPUs: A Fixed-Size Hashed Page Table

Sungbin Jang
Sungkyunkwan University
Republic of Korea
sunbi3361@skku.edu

Junhyeok Park
Sungkyunkwan University
Republic of Korea
vzx00770@skku.edu

Osang Kwon
Sungkyunkwan University
Republic of Korea
osang915@skku.edu

Yongho Lee
Sungkyunkwan University
Republic of Korea
jhyn205@skku.edu

Seokin Hong
Sungkyunkwan University
Republic of Korea
seokin@skku.edu

## Abstract

GPU memory virtualization has become essential for efficient programming, memory management, and address space sharing among computing devices in heterogeneous systems. Conventional GPU virtual memory systems use multi-level Radix Page Tables (RPTs) to store virtual-to-physical address mapping in device (GPU) memory. When a TLB miss occurs, a page table walker accesses each level of the page table sequentially to find the desired mapping. These sequential accesses significantly degrade performance, adding pressure to the GPU memory hierarchy. To make matters worse, recent computing systems now support five-level RPTs, further increasing the number of memory accesses required per page table walk.

To tackle this problem, we propose a novel framework called *Fixed-Size HPT (FS-HPT)*, which employs Hashed Page Tables (HPTs) instead of traditional RPTs. Our framework is motivated by two key observations. First, a GPU's local page table is primarily responsible for storing the Page Table Entries (PTEs) of pages currently in GPU memory. Second, most remote mappings are only live for a short time and account for a small portion of the page table during program execution. Motivated by these observations, FS-HPT uses a large fixed-size hash table as the GPU's local page table. In the proposed framework, the page table size does not grow. Thus, our approach fundamentally avoids page table resizing, a critical limitation of HPTs. Instead, FS-HPT strategically evicts rarely-used PTEs from the page table to reduce hash collisions. FS-HPT employs a *step table* to provide fast table lookups and a *victim buffer* to minimize the impact of PTE eviction on performance. These additional components incur negligible overhead. Our experimental results demonstrate that for irregular memory-intensive workloads, FS-HPT and FS-HPT integrated with the state-of-the-art page table walk technique outperform RPTs by an average of 27.8% and 61.7%, respectively.

*PACT '24, October 14–16, 2024, Long Beach, CA, USA*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676900

## CCS Concepts

• **Computing methodologies → Graphics processors**; • **Software and its engineering → Virtual memory**.

## Keywords

GPGPU, Virtual Memory, Page Tables

**ACM Reference Format:**
Sungbin Jang, Junhyeok Park, Osang Kwon, Yongho Lee, and Seokin Hong. 2024. Rethinking Page Table Structure for Fast Address Translation in GPUs: A Fixed-Size Hashed Page Table. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3656019.3676900

## 1 Introduction

Modern GPUs supports memory virtualization to enable multi-tenancy [38, 45], sharing address space with CPUs [48, 49], and demand paging [1, 27, 47]. However, the benefits provided by memory virtualization do not come for free. To support memory virtualization, GPUs employ hardware components such as a Translation Lookaside Buffer (TLB) and a GPU MMU (GMMU). Before accessing data pages, the GPU must translate a virtual address to a physical address. For this translation, the GPU first accesses the TLB, and when the translation request is missed in the TLB, it looks up a page table. A discrete GPU typically has its own page table in the device memory that maintains the virtual-to-physical mappings. Conventional GPU page tables use a multi-level tree structure known as Radix Page Tables (RPTs) [36, 37].

Due to GPUs experiencing frequent TLB misses, the address translation process can become a significant performance bottleneck. Over the years, many efforts have been made to mitigate these address translation overheads in GPUs. Increasing TLB reach by coalescing TLB entries or page promotion can effectively reduce address translation overhead [6, 29, 42]. Li et al. and Baruah et al. exploited locality within or between Cooperative Thread Arrays (CTAs) to increase the TLB hit rate [8, 31]. Several prior works have attempted to reduce page table lookup overhead as the current GPU page table design (i.e., RPTs) involves sequential pointer-chasing accesses, known as page table walk, which causes severe performance degradation. Shin et al. reduced the TLB miss penalty by scheduling page table walk requests [48] and by coalescing page table walks into a single memory access [49].

However, all these techniques use the inefficient multi-level RPTs as the baseline. Although it is possible to use Page Walk Caches (PWCs) [7, 9, 10, 21] to replace these sequential pointer-chasing operations with an MMU cache lookup, PWCs do not work effectively for emerging memory-intensive workloads with large memory footprints and irregular memory access patterns [41, 50, 54]. To make matters worse, recent computing systems now support five-level RPTs, expanding the virtual and physical address space [24] and further increasing page table walk overheads.

In this paper, we propose a novel framework, called *Fixed-Size HPT (FS-HPT)*, which aims to tackle the fundamental limitation of current RPT-based GPU virtual memory. FS-HPT adopts *Hashed Page Tables (HPTs)* instead of RPTs for the GPU page table. Since HPTs use a hash function to calculate a hash value and directly use this as the table index, the HPT lookup requires only a memory reference (if there is no hash collision). This fast lookup mechanism of HPTs can eliminate the expensive pointer-chasing memory accesses required by conventional RPTs.

Several recent works have proposed HPTs in the CPU domain [50, 51, 54]. State-of-the-art HPT designs [50, 51] reduce address translation overheads by an average of 41%. They elaborately designed the HPTs to effectively manage hash collisions, one of the critical challenges when using HPTs [54]. However, prior HPT designs are primarily optimized for CPUs, so it is not straightforward to simply apply those designs to GPUs. They necessitate adjustments in the page table size to manage hash collisions, which requires migrating Page Table Entries (PTEs) and frequently relies on OS support. The PTE migrations can significantly increase memory allocation time, and frequent OS interventions decrease GPU performance [29]. In addition, state-of-the-art HPT design issues multiple memory requests simultaneously for a page table walk to exploit memory-level parallelism. This can lead to performance degradation since GPUs are more bandwidth-sensitive than CPUs.

We exploit two key observations to efficiently adopt HPTs in GPU's virtual memory subsystems. First, there is an upper bound for the number of PTEs the GPU's page table can hold because the page table is responsible primarily for maintaining the PTEs of pages currently stored in the GPU memory. Second, the number of remote mappings stored in the GPU page table accounts for only a small portion of total entries. Recent GPUs allow direct access to remote pages in the other devices [16, 47, 53]. To support direct remote access, the GPU's page table needs to hold the PTEs for both local and remote pages, which increases the load factor as the page table stores more remote mappings. However, we found that only a few remote mappings will be accessed again because frequently accessed remote mappings become local mappings.

Motivated by these two key observations, the FS-HPT framework employs a fixed-size hashed page table whose size is sufficiently large to mostly avoid the hash collisions, thereby eliminating dynamic resizing. FS-HPT uses open-addressing to handle hash collisions during PTE allocation and stores open-addressing steps in a *step table* to enable fast PTE accesses during address translation. Accessing the step table can incur additional memory references as the table exists in the device's memory. To address this issue, we repurpose the existing MMU cache (PWCs) as a *step cache* for storing recently accessed step table entries. We also propose a *remote PTE eviction policy* to keep the occupancy of the HPT (i.e., load factor) below a target level. With this remote PTE eviction policy, the GPU driver [39] evicts rarely-used remote mappings from the page table and stores them in a *victim buffer* implemented as a radix tree structure for scalability.

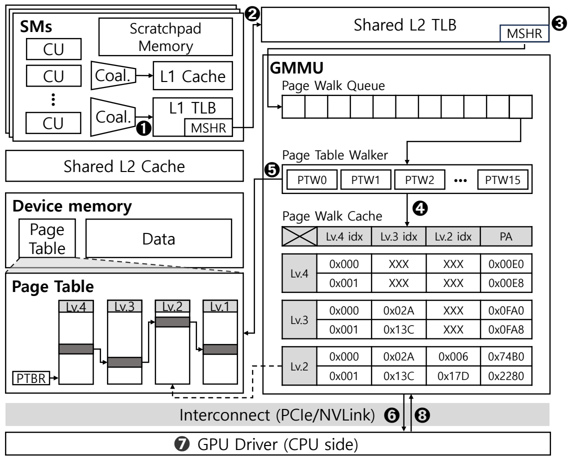


**Figure 1: Baseline GPU architecture**

The main contributions of this paper are as follows:

- We propose a novel framework, called *FS-HPT*, to improve GPU virtual memory. FS-HPT employs a fixed-size hashed page table to enable rapid page table lookup. In most cases, FS-HPT can retrieve page table entries with a single memory reference by using a *step table* and *step cache*. To the best of our knowledge, this paper is the first work to study the effectiveness of using HPTs as the GPU's page table.
- We address a limitation regarding remote mappings by employing a *remote PTE eviction policy* and *victim buffer*.
- We evaluate the performance of FS-HPT using a detailed GPU simulator. For irregular memory-intensive workloads, FS-HPT and FS-HPT integrated with the state-of-the-art page table walk technique outperform RPTs by an average of 27.8% and 61.7%, respectively.

## 2 Background and Motivation

### 2.1 GPU Virtual Memory

Recent GPU systems have adopted the virtual memory [2, 36, 43, 44, 47] to share a virtual memory space between GPUs and CPUs. There are two typical ways to enable shared virtual memory in GPU systems. One is sharing a page table between the CPUs and GPUs via I/O Memory Management Unit (IOMMU) hardware [43, 44]. This approach is typically applied to integrated GPUs that share the main memory with CPUs [48, 49]. The second way is to let CPUs and GPUs manage their own page tables separately. The CPU page tables have the entire information, while the GPUs only retain certain parts of the address translation information in their distinct (local) page tables [32]. This organization is beneficial when GPUs have their own distinct memory (VRAM) with a GMMU since GPUs can then perform address translation without communication with the CPUs (via the IOMMU) [29, 30, 32, 45]. In this paper, we select the latter approach (i.e., address translation using a GMMU and distinct GPU page tables) as the baseline architecture.

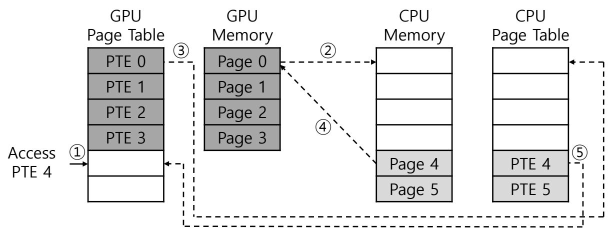


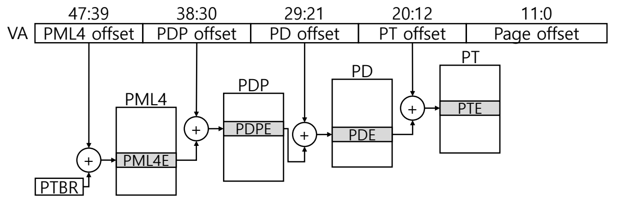
Figure 2: PTE eviction and allocation in GPUs

**Address translation:** Figure 1 depicts the address translation process in GPUs. Each Streaming Multiprocessor (SM) has a private L1 TLB. When the SM issues memory instructions (load/store), it generates address translation requests to get a physical address. Before the requests are forwarded to the L1 TLB, a coalescer merges these requests into a single translation request if they fall into a single page range (4KB, 64KB, or 2MB) (❶). Upon an L1 TLB miss, Miss Status Holding Registers (MSHRs) merge missed requests, and these merged requests are sent to the shared L2 TLB (❷).

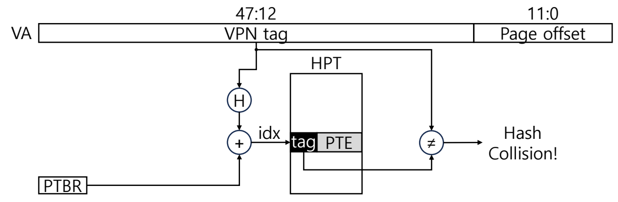
If an L2 TLB miss occurs, it sends the translation requests to the GMMU, which is shared across all SMs (❸). The GMMU enqueues arrived requests into a page walk queue. Then, a page table walker pops the translation requests from the queue, and the walker looks up the page table on the GPU side by sending memory requests to the cache hierarchy (❺). Since all SMs generate multiple address translation requests simultaneously, there are tens of page table walkers in the GMMU to increase parallelism [44]. We use four-level Radix Page Tables (RPTs) [18] as a baseline [30, 37, 45, 48]. To mitigate sequential memory access of RPTs, the GMMU equips PWCs [7] to cache the physical address of an intermediate page table (except the last-level page table). As a result, the page table walker accesses PWCs before sending requests to the cache hierarchy, and if it hits, the walker can skip memory accesses (❹).

**Page fault handling:** Basically, GPUs with the UVM system offer the demand paging scheme [15]. Even though the GPUs have performed the whole address translation process (❶-❺), the address translation information (i.e., the virtual-to-physical mapping) may not exist in the GPU local page table. This situation indicates that the desired page is not in GPU local memory, so the GMMU generates a far fault and sends an interrupt signal to the CPU [1] (❻). On the CPU side, the GPU driver [1] walks the host-side page table to get the required PTE and accesses the data page using that PTE (❼). Then, the GPU driver sends the data page to the GPU's local memory and populates the corresponding PTE in the GPU's page table using the interconnect (PCIe or NVLink [35, 36]). Finally, the address translation is replayed on the GPU side (❽).

**Managing GPU page tables:** The GPU driver on the CPU side manages the GPU's page table before launching the kernel and handling any page faults. Basically, the GPU's page table stores valid page table entries for pages in the device (GPU) memory, and the CPU's page table stores valid PTEs for pages in the host (CPU) memory [39, 47]. Figure 2 depicts how the PTEs are managed between the GPU's and CPU's page tables. When the GPU tries to access PTE 4 for Page 4, the GMMU incurs a page fault because Page 4 is not in the device (GPU) memory (①). Next, the GPU driver migrates Page 4 from CPU to GPU memory. However, in the example shown in the figure 2, there is not enough space in GPU memory, so the driver selects a victim page using a page eviction policy [15]. In the example, Page 0 is evicted from GPU memory and migrated to the CPU (②). Since Page 0 is evicted, the driver invalidates PTE 0 in the GPU's page table and populates the CPU's page table with PTE 0 (③). After the GPU driver migrates Page 4 from CPU to GPU memory (④). Also, the driver invalidates PTE 4 in the CPU's page table and populates the GPU's page table with PTE 4 (⑤). Unlike the CPU, the GPU does not necessarily maintain the PTEs of the pages swapped out to the CPU (invalid PTEs) since the CPU's page table maintains PTEs of swapped-out pages [39].



(a) Radix Page Table



(b) Hashed Page Table

Figure 3: Radix page tables and hashed page tables (a) Radix page tables (RPTs) (b) Hashed page tables (HPTs).

## 2.2 Multi-Level Radix Page Tables

Conventional CPUs and GPUs typically adopt RPTs [18, 21, 23, 24] to maintain virtual-to-physical translation information. Figure 3a depicts a four-level RPT. The page table walk process for four-level RPTs is as follows. The upper 36 bits of a Virtual Address (VA) is the Virtual Page Number (VPN), and this VPN is then divided into four 9-bit fields that represent an index for each of the four levels in the page table. The translation process starts with the Page Table Base Register (PTBR), which stores a Page Map Level-4 (PML4) base address. We can access a PML4 entry, which contains a base address of the Page Directory Pointer (PDP), by adding the value in the PTBR to the upper 9 bits (i.e., VA[47:39]) of the VPN. Now, we have a PDP base address and can get the address of the next level page table (PD) by adding the PDP base address to the next 9 bits (i.e., VA[38:30]) of the VPN. This process continues and ends when the page table walker reaches the PTE, which contains the data page's Physical Page Number (PPN).

As described above, a single memory instruction results in five memory accesses (four accesses for address translation and one access for data). This stalls the core for a long time and adds significant pressure to the GPU memory hierarchy [6, 7, 29, 42, 48, 49, 54]. In the worst case, all five memory accesses on the L2 cache could be missed, resulting in five sequential DRAM accesses.

**Limitations of radix page tables:** Emerging workloads (e.g., graph processing [13, 34], bioinformatics [11, 33], and large language models [52]) show irregular memory access patterns with

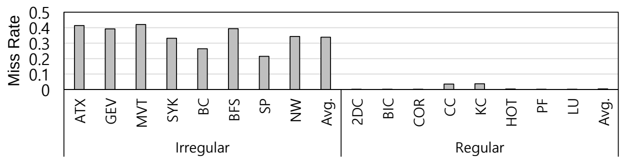


**Figure 4: Miss rate of level-2 PWC**

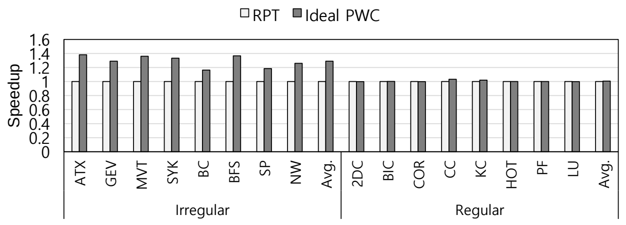


**Figure 5: Performance of four-level RPT and ideal PWC**

large memory footprints. These workloads frequently incur TLB misses, leading to numerous page table walks. In addition, due to the inherent parallelism of GPUs, these workloads typically generate multiple simultaneous page table walk requests, putting significant pressure on the cache hierarchy. [43, 44, 48, 49]. Caching these frequently and simultaneously generated page table walk requests within a small PWC structure is difficult, resulting in more than one memory access per page table walk.

Figure 4 shows the miss rates of a level-2 PWC for various benchmarks. Since each entry in the level-4 and level-3 PWC covers 512GB and 1GB logical region, respectively, the hit rate for these levels is almost 100%. Therefore, we focus on the miss rate of a level-2 PWC. The benchmarks and simulation environment are described in Section 5.1. We categorize the benchmarks based on their level-2 PWC miss rates. Workloads with a level-2 PWC miss rate exceeding 20% are categorized as irregular, showing an average miss rate of 33.8%. In contrast, workloads with a level-2 PWC miss rate of 20% or less are categorized as regular, with an average miss rate of 0.4%. To measure the impact of PWC misses on performance, we compare the performance between a four-level RPT and an ideal PWC configuration, where every page table walk hits at PWCs. Figure 5 shows the speedup of an ideal PWC over RPT. Since irregular workloads suffer from frequent PWC misses, the ideal PWC shows a speedup of 29% over RPT. This result demonstrates that reducing PWC misses can significantly enhance overall performance.

One might suggest that scaling TLBs or PWCs could mitigate page table walk overheads in RPTs. However, since TLBs and PWCs serve as the SRAM cache, they are hard to scale in terms of latency and area [29, 50, 54]. Additionally, with the upcoming five-level RPTs, it becomes more challenging to scale them [24, 28, 50, 51, 54].

## 2.3 Hashed Page Tables

To reduce the substantial overheads associated with address translation using RPTs, previous studies have explored Hashed Page Tables (HPTs). They have been implemented HPTs as an alternative page table structure in some commercial CPUs [18–20, 50, 51, 54] since the HPTs do not need sequential page table walks.

Figure 3b illustrates the basic structure of HPTs. A hash function takes all bits of the VPN as input, and the output (a hash value) is used as an offset of the page table. We can get the physical address of the PTE for the requested page by adding the hash value to the PTBR value. As HPTs can directly generate the PTE's physical address, they can potentially offer faster address translation compared to the RPTs, which may require multiple memory references. Therefore, by using HPTs, we can achieve significant performance gains comparable to the ideal PWCs.

However, due to the inherent nature of hash functions, distinct hash keys (i.e., the input of the hash function) may yield identical hash values, known as a *hash collision*. In a general hash table, a hash key is a unique identifier derived from the processed data. In HPTs, the VPN tag serves as the hash key. For precise translation, we must store the hash key in each hash slot and compare the hash key with the input virtual address's VPN tag. If the hash key and VPN tag do not match, we must handle this collision. Handling hash collisions adds complexities to HPT design, so managing these collisions is one of the key challenges of HPTs.

**Hashed page tables in the CPU domain:** State-of-the-art HPT techniques [50, 51] have been proposed to resolve the hash collision issue efficiently. Skarlatos et al. [50] proposed an Elastic Cuckoo Page Table (ECPT) that prevents frequent hash collisions by resizing the page table during program execution. The ECPT can operate effectively with OS support and outperforms RPTs. However, this approach requires resizing the page table during program execution, which is very cumbersome for GPUs. Resizing the page table necessitates rehashing all its entries, leading to migration of the page table entries and requiring frequent support from the OS. These PTE migrations and frequent OS interventions prolong the time it takes to handle PTE allocation within the GPU domain [1].

## 2.4 Motivation: Why Are HPTs a Good Choice for GPU Page Tables?

**Dynamic resizing of the page table is not necessary:** The reasons are twofold. First, the local page table primarily holds the (local) mappings for the pages stored in GPU memory, allowing a load factor[1] of the hashed page table can be maintained without resizing the table. A lower load factor reduces the likelihood of hash collisions. As described in Section 2.1, since the CPU's page table maintains any evicted PTEs from the GPU, the GPU's page table only needs to store valid PTEs of the pages currently stored in the device memory. Thus, the number of valid PTEs in the GPU's page table reaches a saturation point when the device memory is almost full, obviating the need for resizing the page table.

Second, the number of live remote mappings in the GPU's local page table will become saturated, and thereby, the page table's load factor does not keep increasing even when it stores remote mappings. Recent GPUs support remote access to host memory or peer GPU memory by storing remote mappings in the GPU's local page table [3, 36]. Especially, recent NVIDIA GPUs [36, 38] use a page access counter to prevent page migration of rarely accessed pages. With the access counter, the GPU driver stores remote mappings in the GPU's page table and migrates only the pages whose access counter value reaches a certain threshold. Even if this mechanism effectively avoids the migration of rarely accessed pages [30], it can

[1]The load factor [14] of a hash table is a metric indicating the ratio of occupied entries to the total number of slots available in the table.

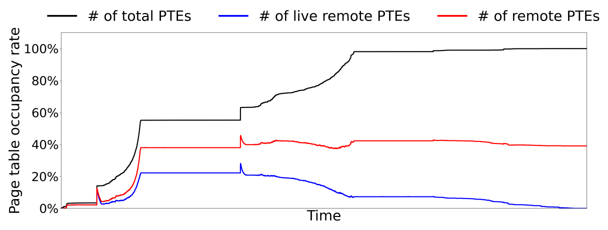


**Figure 6: The number of live remote mappings of BFS [34].**

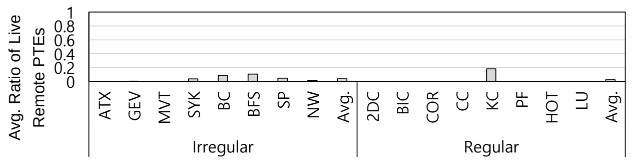


**Figure 7: Average ratio of live remote mappings throughout program execution.**

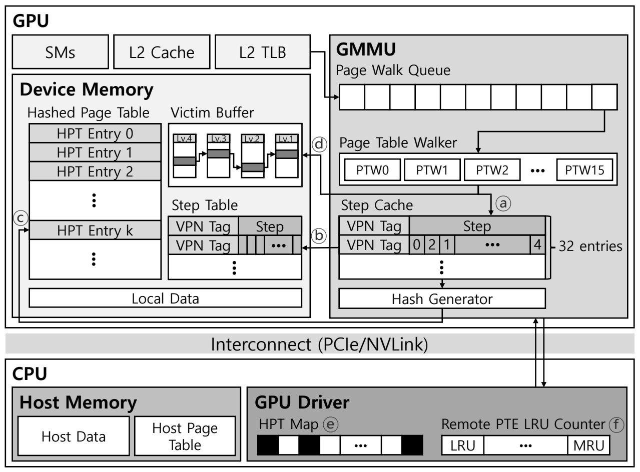


**Figure 8: FS-HPT overview**

increase the load factor of the page table as the GPU must store remote mappings, causing more hash collisions.

However, we have found that a GPU's page table does not need to store all remote mappings. Figure 6 shows the liveness of remote mappings during the execution of BFS [34]. The "liveness" of a remote mapping is defined as the duration from the initial access (first touch) of the remote mapping to its final access (last touch). We measure liveness using the access counter-based page migration [39]. As shown in the figure, the number of live remote mappings decreases during program execution. This is because frequently accessed remote mappings (hot remote PTEs) quickly become local PTEs as the corresponding pages are migrated to the GPU memory, and the infrequently accessed remote mappings (cold remote PTEs) tend to fall into disuse after their initial period of activity. Figure 7 shows the average proportion of live remote mappings relative to the total number of page table entries for all the benchmarks. The ratio of the live remote mappings ranges from 0% to 20% for the various benchmarks (with an average of 0.16%). These results indicate that the number of live remote mappings does not gradually increase in the GPU's page table throughout the execution of the programs.

**The page table occupies a small portion of GPU memory:** For a 4KB base page size, the associated PTE size is eight bytes [37]. Therefore, GPUs that use RPTs require only 0.2% of total GPU memory to store PTEs for pages that reside in GPU memory. If we adopt HPTs for the GPU's page table, we need a larger page table than when using RPTs to avoid hash collisions. Since the page table size for storing all the mappings of the pages in GPU memory accounts for a small portion of the GPU memory, even if we increase the page table size, the area overhead is negligible. For example, the page table, which is 2.5x times larger (for a target load factor of 0.4) than RPTs, only requires 0.5% of total GPU memory.

## 3 Fixed-Size Hashed Page Table

### 3.1 Overview

**Key idea - Adopting HPTs as the GPU's page table structure:** As described in Section 2.2, RPTs incur significant performance degradation, and they also show poor scalability with increasing address space due to the sequential table lookup process required by their hierarchical structure. HPTs are a viable alternative due to their ability to quickly look up the desired page table entry using a hash function. With the observation that GPUs do not need to resize page tables dynamically, we apply HPTs to the GPU memory system using a fixed-size page table. In this section, we present the ***Fixed-Size Hashed Page Table (FS-HPT)***, a novel GPU virtual memory framework that adopts Hashed Page Tables (HPTs) to address the fundamental limitation of the conventional page table structure (i.e., RPTs).

**Challenges and solutions:** In order to adopt HPTs in GPU architectures, we need to address two key challenges. First, hash collisions could occur even if we set a large page table size, so we have to resolve them efficiently. To handle hash collisions, we adopt open-addressing as it is simple to implement in the GPU driver and GMMU at a low cost [14]. Hash collision handling with the open-addressing scheme does lead to multiple memory accesses due to the sequential probing scheme. We tackle this issue with a *step table* and *step cache* (Section 3.3-3.5). With these, FS-HPT can look up the page table with only one memory access.

Second, remote mappings can increase the load factor of the page table. As described in Section 2.4, live remote mappings occupy a tiny fraction of total PTEs. Thus, if we store only hot remote mappings in the page table, we can reduce hash collisions. To this end, we propose a *remote PTE eviction policy*, which strategically evicts rarely-used remote mappings from the page table (Section 3.6). However, if the evicted mapping is used in the future, it will cause a costly page fault. Thus, we also introduce the *victim buffer* to hold evicted remote mappings in the device memory (Section 3.7). The GMMU first looks up the step cache and step table, and if there are no matching entries, it accesses the victim buffer to get the requested remote mappings.

Figure 8 depicts the overall architecture of our FS-HPT framework. The device memory contains the hashed page table, step table, and victim buffer. In the GMMU, we repurpose the remaining PWCs to use as the step cache and add a hash generator. When an L2 TLB miss occurs, the page table walker first accesses the step cache (ⓐ). If there is no entry in the step cache, the walker accesses the step table and fills the step cache (ⓑ). After, the walker looks up

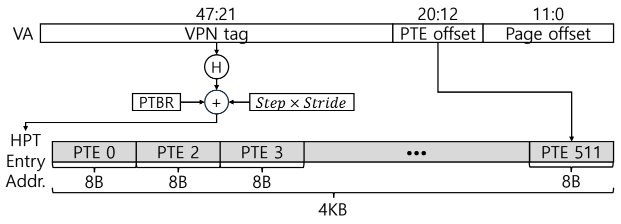


**Figure 9: HPT entry format**

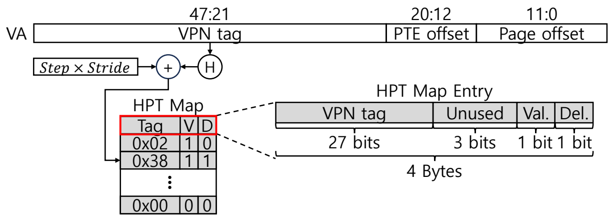


**Figure 10: PTE allocation with HPT Map**

the page table with the value derived from the step table entry (ⓒ). If there is no matching entry in the step table, the walker accesses the victim buffer to get the remote mappings (ⓓ). Finally, if there are no matching mappings in the victim buffer, the GMMU incurs a page fault, and the GPU driver populates the GPU's page table by referencing an HPT Map (ⓔ). If there is no space for storing new PTE, the driver evicts a remote mapping from the page table using a remote PTE LRU counter maintained by the driver (ⓕ).

## 3.2 HPT Entry Format

The naive implementation of HPTs shown in Figure 3b causes some problems. First, there will be a notable decrease in the spatial locality of the PTEs [50, 54]. This is because allocating a PTE for each HPT entry does not ensure the spatial locality of PTEs in cacheline granularity. Second, since the GPU driver evicts pages with a 2MB Virtual Address Block granularity [1, 27], evicting pages from the GPU incurs 512 lookups in the hashed page table to remove the PTEs for the evicted pages. Finally, it is difficult to support various page sizes simultaneously because the number of address bits used as a VPN tag varies depending on page sizes. For example, if a GPU uses 4KB and 2MB page sizes at the same time, the GMMU has to access the page table twice with different VPN tags: VA[47:12] for the 4KB page size and VA[47:21] for the 2MB page size. This dual-access approach is necessary because the page size cannot be determined with only a given VA.

To address these problems, we propose the new HPT entry format. Figure 9 shows a proposed HPT entry format. We cluster 512 PTEs of contiguous pages in a single hash table entry so that each entry represents a contiguous 2MB memory region. The PTEs in an HPT entry are indexed with a PTE offset obtained from the VA[20:12]. This design ensures locality among PTEs and requires only a single HPT lookup during page eviction. In addition, with this HPT entry format, the VPN tag remains the same across all supported page sizes (i.e., 4KB, 64KB, and 2MB [37]). Therefore, the virtual address is translated using only one access to the table, even when various page sizes are supported in the GPU virtual memory.

## 3.3 Hash Collision Resolution

There are several methods to resolve hash collisions, such as chaining [14], open-addressing [14], and cuckoo hashing [40]. Among these, we employ the open addressing scheme, which is simple and thus can be implemented effectively in the GMMU and the GPU driver. When inserting a new key-value pair, a hash function calculates an index based on the key. If the entry associated with the calculated index is empty, the key-value pair is inserted. If the entry is already occupied by a different key (indicating a collision), the open addressing scheme seeks an alternative empty entry for the key using a specific probing sequence. In our implementation, the table index of the next candidate entry for the key is calculated using *Stride* and *Step*. During the iterative probing for an empty entry, the driver increments the *Step* until the empty one is found ($new\ index = hash\ value + Stride \times Step$). The *Stride* is pre-determined by the GPU driver when creating the page table.

## 3.4 PTE Allocation

When the GPU requires data pages that are not currently in GPU memory, it causes a page fault, and the GPU driver on the CPU side allocates the appropriate PTEs and migrates the data pages to the GPU. To deal with PTE allocation, we introduce a software-defined data structure called the *HPT Map* in the driver. The HPT Map maintains allocation information on the HPT stored in device memory; each HPT entry has a corresponding entry in the HPT map. As illustrated in Figure 10, an HPT Map entry consists of a VPN tag, a valid bit, and a deleted bit. The VPN tag is used to verify whether the accessed entry is the correct one for the given VA, and the valid bit indicates whether the corresponding HPT entry is valid. The deleted bit indicates that the corresponding HPT entry is no longer valid and is part of a probing sequence. In open addressing, simply deleting an entry and leaving it empty (invalid) can disrupt the continuity of the probing sequence. The deleted bit simply addresses this challenge. Note that the driver can reuse the deleted entry to allocate a new HPT entry because its valid bit is invalid. In other words, if there is a deleted entry before finding an invalid entry, the driver allocates a new HPT entry in the same place as the deleted entry.

Algorithm 1 describes the PTE allocation process. First, the GPU driver accesses the HPT Map entry, which is pointed to by the index calculated by adding the hash value and HPT Map base address (line 4). If the accessed entry is valid, the driver compares the new PTE's VPN tag with the tag in the accessed HPT Map entry (lines 5-6). If the tag matches, the driver allocates a PTE at the existing HPT entry (line 7). Otherwise, the driver increments the *Step* and accesses the next entry of the HPT Map (line 10). If the deleted bit of the accessed entry is set, the driver increments the *Step* and accesses the next entry of the HPT Map (line 16). To reuse a deleted entry, the driver keeps the address of the first accessed deleted entry (lines 13-14). If the accessed entry is invalid, it means there are no allocated HPT entries for the requested VA. Then, the driver checks whether oversubscription has occurred or if the number of HPT entries has reached the threshold to guarantee enough space for the data page and PTEs. If oversubscription occurs, the driver evicts a local HPT entry (lines 18-19). If the number of HPT entries reaches the threshold, the driver evicts a remote entry (lines 20-21). Remote PTE eviction will be described in the section 3.6. Before

**Algorithm 1** PTE Allocation

```
 1: $Step \leftarrow 0$
 2: $Del\ Addr \leftarrow null$
 3: while $true$ do
 4:     $Current\ Addr \leftarrow Base\ Addr + Hash\ Value + (Step \times Stride)$
 5:     if $Valid$ then
 6:         if $VPN\ tag == Tag$ then
 7:             Allocate PTE                          ▷ HPT entry already exists
 8:             break
 9:         else
10:             $Step \leftarrow Step + 1$                     ▷ Tag mismatch
11:         end if
12:     else if $Deleted$ then
13:         if $Del\ Addr == null$ then
14:             $Del\ Addr \leftarrow Current\ Addr$     ▷ Save first deleted entry address
15:         end if
16:         $Step \leftarrow Step + 1$                         ▷ Deleted entry
17:     else if $not\ Valid$ then
18:         if Oversubscription then
19:             Page eviction
20:         else if $Current\ HPT\ entries \geq Max.\ HPT\ entries$ then
21:             Remote PTE eviction
22:         end if
23:         if $Del\ Addr != null$ then
24:             $Current\ Addr \leftarrow Del\ Addr$             ▷ Reuse deleted entry
25:         end if
26:         $Valid \leftarrow true$                          ▷ Allocate new HPT entry
27:         Allocate PTE
28:         break
29:     end if
30: end while
```

allocating a new HPT entry, the driver replaces the current address with the deleted address if it exists (lines 23-24). Finally, the driver allocates a new HPT entry and a new PTE (lines 26-27).

## 3.5 PTE Access with Step Table

**Step table:** The page table walker can access the HPT with a single memory reference if there are no hash collisions. However, if hash collisions occur, the walker needs to perform a probing sequence, necessitating multiple sequential memory accesses to the HPT. To tackle this issue with open addressing, we employ a *step table*. This is designed to store the open-addressing steps for each HPT entry. When the GPU driver populates the GPU's page table, the driver also stores the *Step* value in the step table. The step table resides in device memory and is implemented as a hash table.

Figure 11 shows the design of the step table. A step table entry stores a tag (VA[47:25]) and open-addressing steps for 16 HPT entries in a contiguous 32MB region. We limit the maximum open-addressing step value to eight, so each step table entry comprises 23 bits for the tag and an additional 48 bits (16×3 bits) for the step values of 16 HPT entries. This setup guarantees that every entry in the step table can represent a 32MB region using just 9 bytes of information. Given the compact size of each entry in the step table, we configure the step table's size to achieve a low load factor of 0.01. This notably diminishes the frequency of hash collisions in the step table. Despite maintaining this low load factor, a step table of this size requires only 0.001% of device memory capacity.

However, when using the step table, the page table walker accesses the memory twice for every HPT lookup: once for the step table and once more for the HPT. To address this issue, we repurpose the existing PWCs as a *step cache* to store recently accessed step table entries. As illustrated in Figure 11, the step cache is designed as a direct-mapped cache structure. It has 32 entries, each comprising an 18-bit tag derived from VA[47:30] of the virtual address

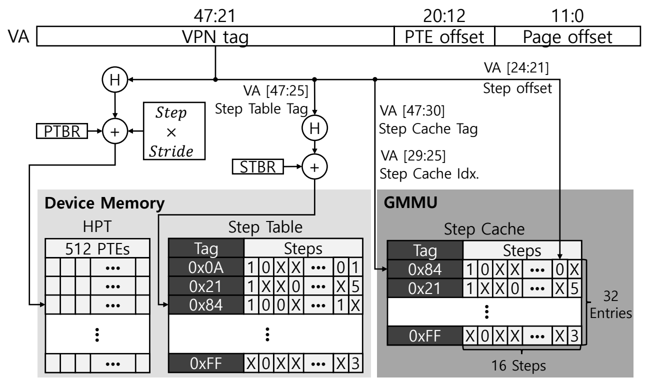


**Figure 11: PTE access with step table**

and 16 step values (48 bits = 3×16). Each entry of the step cache is designated for 16 HPT entries allocated to a 32MB contiguous region. As a result, the page table walker accesses the step cache before accessing the step table. Since the step cache covers a 1GB region, its hit rate is nearly 100%, ensuring that PTEs are mostly retrievable with only a single memory access. It is worth noting that the step cache substantially outperforms PWCs in terms of area efficiency as it covers a 2MB region only using 3 bits, while the PWC covers a 2MB region using 64 bits.

**PTE access:** Figure 11 also shows the page table access process. For each page table walk request, the page table walker first looks up the step cache. If there is no matching entry in the step cache, the walker accesses the step table with an index calculated by adding the value in the Step Table Base Register (STBR) and the hash value from VA[47:25]. Next, it fills the step cache. If there is no matching entry in the step table, the GMMU incurs a page fault. Otherwise, if there is a matching entry, the walker accesses the HPT by adding the value in the Page Table Base Register (PTBR), the hash value from VA[47:21], and the $Step \times Stride$. The GPU driver pre-determines the $Stride$ value, while the $Step$ value is derived from the step cache. Finally, the walker gets the requested PTE from the HPT entry using the PTE offset (VA[20:12]) field.

## 3.6 Remote PTE Eviction Policy

Remote mappings can mitigate page migration overheads and page thrashing issues in discrete GPU systems [16, 30, 47, 53]. However, if the number of remote mappings stored in the GPU's page table increases, its load factor can also keep increasing, leading to frequent hash collisions in the page allocation process. As described in Section 2.4, live remote mappings occupy only a small fraction of the total page table capacity during program execution. Consequently, by storing only the live (or hot) remote mappings in the page table, we can effectively prevent the load factor from continually increasing.

Figure 12 shows the concept of the remote PTE eviction policy we propose to constrain the load factor of FS-HPT. The upper side of the figure shows the page eviction policy current GPU drivers already use [15, 39]. When oversubscription occurs, the GPU driver selects a victim page from the front of the Least Recently Used (LRU) list and evicts it (lines 18-19 in Algorithm 1). The LRU list is updated for every time a page fault is handled.

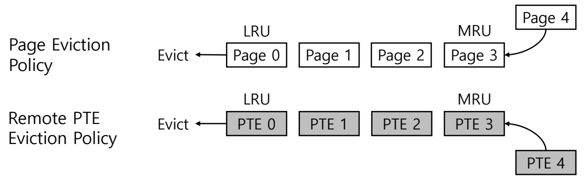


**Figure 12: Page replacement and remote PTE eviction**

We extend this replacement policy to evict remote mappings from the GPU's page table. The lower side of the figure shows the proposed *remote PTE eviction policy*. When allocating a new HPT entry, if the load factor of the GPU's page table reaches the designated *load factor threshold*, the driver selects a victim entry front of the LRU list and evicts that entry from the page table. The GPU driver can update the LRU list since the GMMU tracks the access frequency of each remote mapping [16, 47, 53]. Also, the following equation (1) shows the calculation to determine the *load factor threshold* used, where $m$ is a multiplicative factor. For example, if there is no oversubscription, we simply set $m$ as 1.0, and the page table can accommodate all required PTEs. On the other hand, if there is an oversubscription, we can adjust the $m$ to be greater than 1.0. If $m$ is 1.2, the page table can store PTEs of pages that cover 120% of the device's memory size. FS-HPT sets the default $m$ as 1.2, but $m$ can be adjusted in some cases. We evaluate the effects of varying $m$ in Section 5.3.

$$load\ factor\ threshold = m \times \frac{\#\ max.\ local\ HPT\ entries}{\#\ total\ HPT\ entries} \quad (1)$$

## 3.7 Victim Buffer

With our remote PTE eviction policy, the GPU driver invalidates remote mappings evicted from the GPU's page table. However, if an evicted remote mapping is accessed again, a page fault occurs. In this case, the GPU driver populates the page table with that mapping again. Even if page faults caused by re-accessing the previously evicted remote entries are infrequent, when they occur, the GPU undertakes costly page fault handling [1].

To minimize the performance impact of the remote PTE replacement, we propose a victim buffer that maintains the evicted remote mappings in device memory. With the victim buffer, we can prevent page faults caused by re-accessing previously evicted remote mappings. Since the number of remote mappings evicted from the HPT (primary page table) varies across applications, we design the victim buffer to be more scalable by using a radix tree structure. While this radix-tree design can lead to sequential memory accesses, it offers significant benefits in managing the victim buffer. Furthermore, the victim buffer is rarely accessed because the hot remote mappings mostly remain in the HPT. Therefore, the performance impact of sequential accesses to the buffer is trivial (Section 5.3).

## 4 Discussion

## 4.1 Step Cache-Less Design

Our design uses an MMU cache, called the step cache, to store open-addressing steps of recently accessed HPT entries. While this design is similar to PWCs, the step cache is more efficient than PWCs as it stores steps instead of physical addresses of the PTEs.

Even if a step cache is indeed more compact than PWCs, it still experiences scaling issues as the application's memory footprint grows. In fact, FS-HPT can operate without the step cache if the page table size is sufficiently large so that hash collisions rarely occur. If hash collisions do not occur in the page allocation process, the *Step* value will be zero for all HPT entries, eliminating the need to read the *Step* value from the step table.

By exploiting this insight, we propose an alternative design, *Step Cache-Less* FS-HPT. This approach excludes the step cache in the GMMU while using a large HPT. Even if this approach increases the area overhead of the page table, the impact of this area overhead is trivial because the page table still basically occupies a small portion of the device memory, as discussed in section 2.4.

To implement the step cache-less design, we need a small modification to FS-HPT. In that design, a hash collision can occur while accessing the HPT. This is because a home HPT entry, whose index is $hash\ value + Stride \times 0$, is always accessed first as we assume that the *Step* is zero for the given VA. We store a VPN tag in each HPT entry to handle hash collisions. Since there are enough unused bits in a PTE [35, 40, 54], we can store the VPN tag in each HPT entry.

If the requested translation is found in the home HPT entry (no hash collision), the page table walker finishes the lookup with only one memory reference. Otherwise, the page table walker reads a *Step* value from the step table and then accesses the page table again using the calculated index. Even if the step cache-less design incurs three memory references per page table walk on a hash collision, it achieves similar performance to the step cache-based design because hash collisions are rare when using a larger HPT. The impact of step cache-less design will be discussed in Section 5.4.

## 4.2 Implementation Overheads

**Hardware overheads**: FS-HPT requires a small amount of additional memory space and a few minor changes in the GMMU unit. The hashed page table accounts for 0.5% of device memory, while a corresponding step table with a maximum load factor of 0.01 occupies 0.001% of device memory. The memory space requirements for the victim buffer vary depending on the memory access patterns of the applications, but it can be freely resized since it uses a multi-level radix tree structure. Also, Employing the step cache incurs no area overhead compared to the baseline architecture because we repurpose existing PWCs as the step cache.
**Software overheads**: FS-HPT requires small changes to the GPU driver. To support a GPU with an 80GB memory [38], the HPT Map occupies only 200MB of memory space in the host memory. Since the GPU driver spends about 80% of page fault handling time for page migration and page unmapping (invalidating) from host page table [1, 27], the time spent for HPT Map searching is negligible. Our remote PTE eviction policy requires maintaining an LRU list of remote mappings. The GPU driver can maintain and update this LRU list by exploiting an existing Access Counter [16, 39, 47].

## 5 Evaluation

## 5.1 Experimental Environment

**Simulation Setup**: To evaluate the proposed technique, we used GPGPU-sim v4.0 [26] configured as similar to an RTX 3070 GPU [38]. The detailed experimental setup is summarized in Table 1.

Table 1: Experimental Setup

| Component | | Parameter |
|---|---|---|
| # of SMs | | 46 |
| Clock frequency | | 1500MHz |
| L1 TLB (private) | | 32 entries, fully associative, 20 cycles, 32 MSHR entries |
| L2 TLB (shared) | | 1024 entries, 16 ways, 80 cycles, 128 MSHR entries |
| L1 cache | | 128KB per SM, 40 cycles |
| L2 cache | | 4MB, 16 ways, 180 cycles |
| Memory | | GDDR6, 16 channels, 448GB/s bandwidth |
| RPT | Structure | 4-level radix page table |
| | Page walk cache | 32 entries, per-level, 4 cycles |
| HPT | Hash collision | Open addressing |
| | Step cache | 32 entries, 4 cycles |
| | Page table size | 0.5% of working set size |
| Page table walker | | 16 page table walkers |
| Page prefetching | | 16 consecutive pages (64KB) |

We faithfully extended the simulator to support RPT-based address translation by implementing multilevel TLBs, page table walkers, page walk caches, and a radix page table. In our simulated GMMU, we modeled highly threaded page table walkers [44] and PWCs with 32 entries [7]. We also modeled the behavior of FS-HPT in the simulator by adding the step table, step cache, victim buffer, and remote PTE eviction policy. We do not consider demand paging overhead in the same way as related works [22, 25, 29] since *FS-HPT* focuses on reducing page table walk overhead. To determine the size of the page table, we assumed that GPU memory capacity is equal to the working set size of the benchmarks. Based on this assumption, we configured the HPT size to cover a memory space that is 2.5 times larger than the benchmark's memory footprint. Setting up this way, the HPT consumes only 0.5% of the GPU memory. We modeled a prefetching scheme that prefetches 16 consecutive pages to the GPU memory when a page fault occurs [15].

**Benchmarks**: We selected diverse benchmarks from Graphbig [34] (BC, BFS, SP, CC, KC), Rodinia [12] (NW, HOT, PF) and Polybench [17] (ATX, GEV, MVT, SYK, 2DC, BIC, COR, LU). The benchmarks exhibiting a level-2 PWC miss rate greater than 20% are categorized as irregular, while those with miss rates lower than 20% fall into the regular category. This classification enables a comprehensive analysis of the proposed technique's efficacy across diverse workloads. The memory footprint of the benchmarks is ranging from 864MB to 2306MB. To mitigate the issue of impractically long simulation times tied to large memory footprints, we constrained the working set size of the benchmarks to between 214 and 1024MB.

## 5.2 Performance Analysis

**Performance comparison**: Figure 13 shows the relative performance of four-level RPT, FS-HPT, and ideal PWC. For the irregular benchmarks, FS-HPT achieves a speedup of 15.8-35.8% (with an average of 27.8%) compared to the four-level RPT. This significant performance improvement with FS-HPT is comparable to the ideal PWC, where we assume all page walks involve only a single memory access. Typically, as the PWC miss rate increases, the performance gain also increases. For example, BC and SP, which have relatively low PWC miss rates compared to other irregular workloads, show speedups of 15.8% and 18.3%, respectively. On the other side, ATX, GEV, MVT, and BFS, whose PWC miss rates are about 40%, achieve

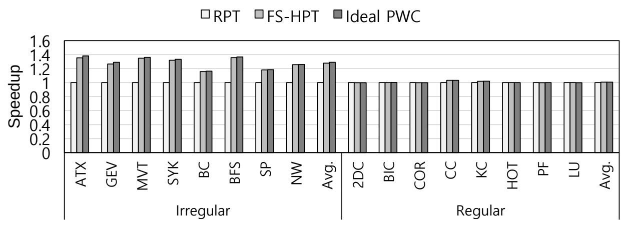


Figure 13: Speedup of FS-HPT over four-level RPT

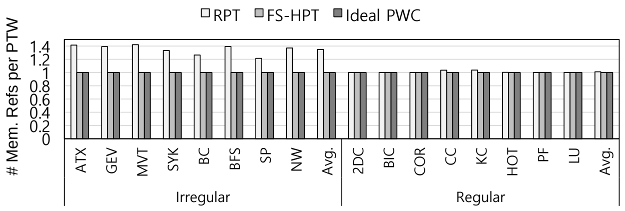


Figure 14: Number of memory references per page table walk

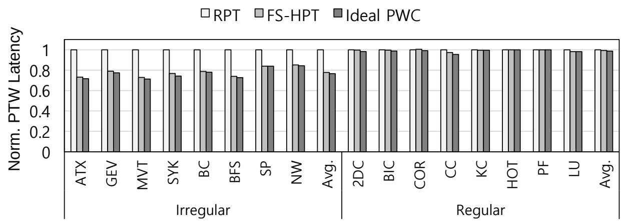


Figure 15: Average page table walk latency.

speedups of 35.4%, 26.6%, 34.9%, and 35.8%, respectively. For the regular workloads, performance is comparable across RPT, FS-HPT, and ideal PWC because most page walk requests hit in the PWC.

**The number of memory references per page table walk**: To understand why FS-HPT offers a significant speedup comparable to an ideal PWC, we examine the behavior of the page table walk process. Figure 14 shows the number of memory references per page table walk for each benchmark running with RPT, FS-HPT, or ideal PWC configuration. For all benchmarks, FS-HPT involves much fewer memory references per page table walk than RPT. For irregular workloads especially, the number of memory references per page table walk for RPT is an average of 1.35, while for FS-HPT it is only 1.01, which is very close to the ideal case.

Since FS-HPT effectively resolves hash collisions with the step table and step cache, it can achieve fewer memory references per page table walk. The step cache achieves an over 99% hit rate for all benchmarks, while conventional PWCs offer only a 65% hit rate on average for irregular benchmarks. This is because an entry of the step cache covers a region 16 times larger (32MB) than an entry of the PWCs. This high hit rate of the step cache makes the average number of memory references per page table walk almost one.

**Average page table walk latency**: Figure 15 shows the average page table walk latency for each benchmark running on RPT, FS-HPT, and ideal PWC. All results are normalized to RPT. For irregular workloads, FS-HPT shows a 22.2% reduction on average in page table walk latency compared to RPT, which is close to the performance of an ideal PWC (23.5%). Since FS-HPT reduces the number of memory references per page table walk, 1) the overall queueing latency for a page walk and 2) the memory access latency per page

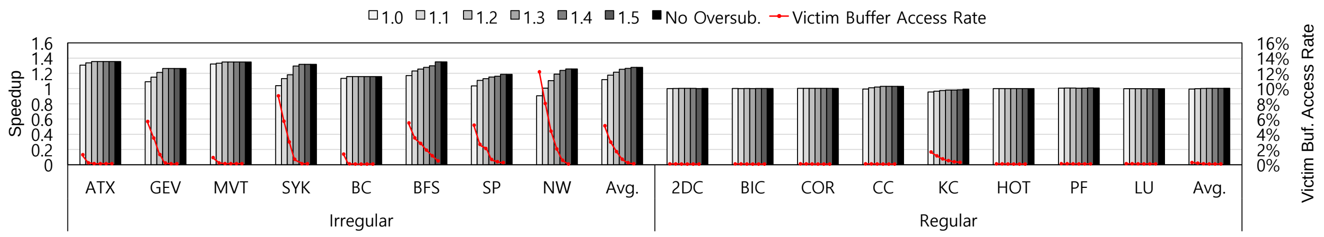


**Figure 16: Speedup of FS-HPT over RPTs with 150% oversubscription varying multiplicative factor**

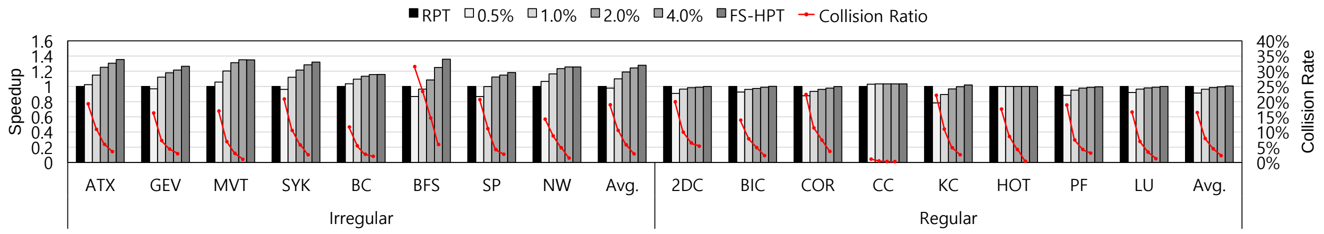


**Figure 17: Performance of Step Cache-less design**

table walk queue both decreases. As a result, the average page table walk latency decreases, reducing the stall time for each SM due to address translation.

## 5.3 Impact of Victim Buffer

We measure the performance of all benchmarks running at 150% oversubscription to evaluate the impact of the victim buffer. For 150% oversubscription, we set the device memory size to 66.6% of each benchmark's working set size to avoid extremely long simulation times. Figure 16 shows the speedup of FS-HPT over RPT both with and without 150% oversubscription, respectively, varying the multiplicative factor $m$ (bar graph). As mentioned in Section 3.6, the default $m$ is 1.2. With $m = 1.2$, for irregular workloads, FS-HPT offers slightly lower performance gains in 150% oversubscription scenarios compared to the case with no oversubscription. The average speedup with $m = 1.2$ is 21.5%, while the average speedup is 27.8% without oversubscription. The small reduction in performance improvement is mainly due to oversubscription requiring accessing of the victim buffer to retrieve cold remote mappings.

Figure 16 also shows the victim buffer access rate as a proportion of total page table accesses for all workloads (line graph). With $m = 1.2$, GEV, SYK, BFS, SP, and NW show over 1% victim buffer access rates (1.4-4.4%). Nevertheless, NW, which has the highest victim buffer access rate (4.4%), still shows a speedup of 11% over RPT. These results demonstrate that while the performance gain from using FS-HPT is slightly reduced with oversubscription, it still provides a substantial performance gain compared to RPT.

**Sensitivity analysis:** We evaluate the performance of FS-HPT under condition of 150% memory oversubscription by varying the multiplicative factor to assess the impact of different *load factor thresholds*. For each multiplicative factor (1.0, 1.1, 1.2, 1.3, 1.4, and 1.5), FS-HPT achieves average speedups of 11.7%, 17.7%, 21.5%, 25.3%, 26.6%, and 27.8%, over RPT, respectively. With $m = 1.0$, all remote accesses result in victim buffer lookups after the load factor reaches its threshold. Therefore, the average speedup over RPT with $m = 1.0$ is 11.7%, which is smaller than in the no oversubscription case. Some benchmarks (NW and KC) show lower performance than RPT with this multiplicative factor because of the high victim buffer access rate. On the other hand, with $m = 1.5$, the GPU's page table can store all the PTEs of pages in the working set with 150% oversubscription. Therefore, the victim buffer access rate becomes zero, and FS-HPT achieves comparable performance to in the no oversubscription case. Since the probing time for open addressing accounts for a small portion of the long page fault handling times, the impact of increasing the multiplicative factor on performance is trivial. Rather, a high multiplicative factor reduces the number of accesses to the victim buffer, providing near-ideal performance.

## 5.4 Performance of Step Cache-Less Design

We evaluate the performance of the step cache-less design discussed in Section 4.1. Figure 17 shows the performance of the step cache-less design and its hash collision rate for page table access with varying sizes of the page table. Each bar graph depicts the speedup of FS-HPT with the step cache-less design for different page table sizes compared to RPT. The line graph shows the hash collision rate on page table lookup. Starting from the base page table size (0.5% of total memory), which is our default configuration, we gradually increase the size of the page table to 1%, 2%, and then 4% of device memory. As expected, while accessing the HPT, the hash collision rate decreases as the page table size increases. With the 0.5% configuration, the performance decreases by an average of 2.2% compared to RPT due to the high collision rate. However, the step cache-less design with page table sizes of 1%, 2%, and 4% of device memory shows an average 10%, 19%, and 24.4% speedup over RPTs for irregular workloads, respectively. These results imply that using a large HPT can achieve near-ideal performance without the need for an SRAM cache in the GMMU.

## 5.5 Impact of Large Page Size

Figure 18 shows the speedup for benchmarks running on RPT, FS-HPT, and ideal PWC over RPT with a 64KB base page size. For irregular workloads, FS-HPT achieves a speedup of 24.5% over RPT,

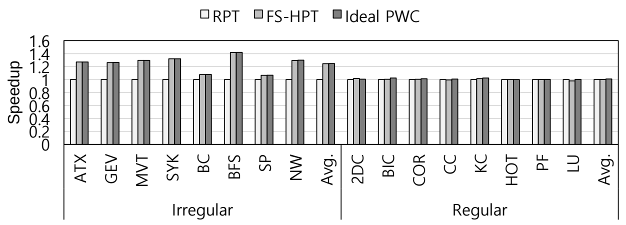


**Figure 18: Speedup of FS-HPT over RPT for 64KB page size**

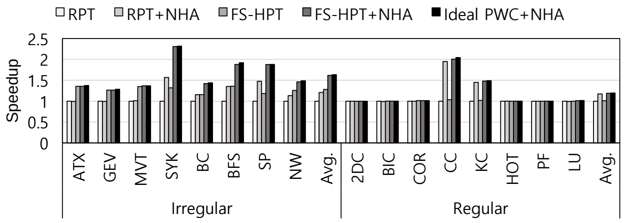


**Figure 19: Speedup of FS-HPT combined with a state-of-the-art page table walk optimization over RPT.**

which is a slight decrease compared to the 4KB page size (27.8%). Since using a large page size increases the TLB reach and thus reduces the number of page table walks, the performance gain of FS-HPT is slightly reduced. However, using a large page is not a panacea since it consumes more PCIe bandwidth [6]. For irregular workloads, using large pages causes internal fragmentation issues because certain workloads may use only a small portion of the required pages. Moreover, FS-HPT is expected to deliver performance enhancements even with large pages, as current large pages may become small pages for emerging workloads.

## 5.6 Comparison and Compatibility with Other Techniques

We compare the performance of FS-HPT with a state-of-the-art technique [49] called Neighborhood-Aware Address Translation (NHA). Figure 19 shows the speedups of RPT, RPT with NHA, FS-HPT, FS-HPT with NHA, and ideal PWC with NHA over RPT. FS-HPT shows an average speedup of 5.8% over RPT with NHA for irregular workloads. RPT with NHA shows no speedup for ATX, GEV, and MVT since there is non-spatial-locality among the page table walks. On the other hand, RPT with NHA gets a speedup for some regular workloads (CC and KC). This is because, unlike FS-HPT, RPT with NHA can reduce the number of page table walks by coalescing the page table walks that fall into the same cacheline.

The strength of FS-HPT comes from its orthogonality to other techniques. Since FS-HPT guarantees the locality of PTEs within 2MB granularity, FS-HPT can be applied with NHA. FS-HPT with NHA shows an average speedup of 61.7% over RPT, which is near the performance of ideal PWC with NHA.

## 6 Related Works

**Optimizing TLB**: There are several works [6, 29, 42] focused on increasing TLB reach by merging TLB entries and using large page [4]. Baruah et al. [8] tried to increase the TLB hit rate by exploiting locality among thread blocks by probing another L1 TLB and by prefetching L1 TLB entries. These techniques could be applied with FS-HPT because our technique focuses on reducing page table walk overheads and can support multiple page sizes.

**Reducing page table walk overhead**: Shin et al. [48] proposed an efficient page table walk scheduling policy by exploiting SIMD execution characteristics. They scheduled page table walk requests using the "shortest-job-first" principle and "batch" processing. Also, Shin et al. [49] reduced the page table walk overhead by coalescing multiple page table walk requests that fall into a single cacheline. Since FS-HPT ensures the spatial locality of PTEs in cacheline granularity, coalescing page table walk requests is compatible with FS-HPT.

**Virtual memory in multi-chip systems**: Pratheek et al. [46] tried to mitigate address translation overhead in Multi-Chip-Module GPUs (MCM GPUs) [5, 55, 56] by exploiting the aggregated capacity of L2 TLB while reducing remote accesses, which causes the NUMA effect. In MCM designs, handling multiple memory references during a page table walk can significantly increase the occurrence of remote accesses. From this perspective, our FS-HPT design is a promising solution, as it offers substantial benefits for future multi-chip GPUs.

## 7 Conclusions

In this paper, we rearchitect GPU virtual memory to tackle its current address translation overheads by replacing multi-level Radix Page Tables (RPTs) with Hashed Page Tables (HPTs). We discovered that GPU page tables primarily store translation information for pages in device memory. Thus, these page tables do not continue to grow without bounds and remain substantially smaller than the device memory's capacity. Leveraging this characteristic, we introduce a Fixed-Size Hashed Page Table (FS-HPT). FS-HPT does not dynamically increase its page table size and thus avoids any overheads from resizing the page table, avoiding a critical limitation in HPTs. Instead, FS-HPT strategically evicts rarely-used Page Table Entries (PTEs) to maintain a target load factor. FS-HPT employs a *step table* to provide fast table lookups by avoiding sequential probing operations when the open addressing scheme is used for collision resolution. To minimize the performance impact of PTE eviction, FS-HPT uses a *victim buffer*, which is slow but scalable, to hold evicted PTEs. Our experimental results show that FS-HPT achieves a 27.8% performance improvement on average over RPT for irregular memory-intensive workloads.

## Acknowledgments

This work was partly supported by the National Research Foundation of Korea (NRF) grant (No. 2022R1C1C1012154, 80%), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-00863, 10%), the Ministry of Science and ICT (MSIT) under the Information Technology Research Center (ITRC) support program (IITP-2021-0-02052, 10%) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP), Samsung Electronics Co., Ltd (IO220902-02337-01), and the BK21 FOUR Project. Sungbin Jang, Junhyeok Park, Yongho Lee, Osang Kwon, and Seokin Hong are with the Department of Electrical and Computer Engineering at Sungkyunkwan University. Seokin Hong is the corresponding author.

## References

[1] Tyler Allen and Rong Ge. 2021. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[2] AMD 2012. *AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE*. AMD. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[3] AMD 2015. *AMD APP SDK OpenCL Optimization Guide*. AMD. https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/programmer-references/AMD_OpenCL_Programming_Optimization_Guide2.pdf

[4] Andrea Arcangeli. 2010. Transparent Hugepage Support. In *KVM forum*, Vol. 9.

[5] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 320–332.

[6] 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*. 136–150.

[7] Thomas W Barr, Alan L Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 48–59.

[8] Trinayan Baruah, Yifan Sun, Saiful A Mojumder, José L Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 455–466.

[9] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. 26–35.

[10] Abhishek Bhattacharjee. 2013. Large-Reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 383–394.

[11] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie Kim, Nika Mansouri Ghiasi, et al. 2022. SeGraM: A Universal Hardware Accelerator for Genomic Sequence-to-Graph and Sequence-to-Sequence Mapping. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 638–655.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. IEEE, 44–54.

[13] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 257–266.

[14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. MIT press.

[15] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between Hardware Prefetcher and Page Eviction Policy in GPU-GPU Unified Virtual Memory. In *Proceedings of the 46th International Symposium on Computer Architecture*. 224–235.

[16] 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, 451–461.

[17] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-Tuning a High-Level Language Targeted to GPU Codes. In *2012 innovative parallel computing (InPar)*. IEEE, 1–10.

[18] Jerry Huck and Jim Hays. 1993. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th annual international symposium on computer architecture*. 39–50.

[19] IBM. 2005. *PowerPC Microprocessor Family: The Programming Environments Manual for 64 and 32-Bit Microprocessors*.

[20] Intel. 2010. Intel Itanium Architecture Software Developer's Manual. (2010). https://www.intel.sg/content/dam/doc/manual/itanium-architecture-vol-1-2-3-4-reference-set-manual.pdf

[21] Intel. 2023. Intel 64 and IA-32 Architectures Software Developer's Manual. (2023). https://cdrdv2.intel.com/v1/dl/getContent/671200

[22] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. Ducati: High-Performance Address Translation by Extending TLB Reach of GPU-Accelerated Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–24.

[23] Corbet Jonathan. 2005. Four-Level Page Tables Merged. (2005). https://lwn.net/Articles/117749/

[24] Corbet Jonathan. 2017. Five-Level Page Tables. (2017). https://lwn.net/Articles/717293/

[25] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 66–78.

[26] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486.

[27] 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*. 1357–1370.

[28] Osang Kwon, Yongho Lee, and Seokin Hong. 2022. Pinning Page Structure Entries to Last-Level Cache for Fast Address Translation. *IEEE Access* 10 (2022), 114552–114565.

[29] Jiwon Lee, Ju Min Lee, Yunho Oh, William J Song, and Won Woo Ro. 2023. SnakeByte: A TLB Design with Adaptive and Recursive Page Merging in GPUs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1195–1207.

[30] Bingyao Li, Yanan Guo, Yueqi Wang, Aamer Jaleel, Jun Yang, and Xulong Tang. 2023. IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1163–1177.

[31] Bingyao Li, Yueqi Wang, and Xulong Tang. 2023. Orchestrated Scheduling and Partitioning for Improved Address Translation in GPUs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[32] Bingyao Li, Jieming Yin, Anup Holey, Youtao Zhang, Jun Yang, and Xulong Tang. 2023. Trans-FW: Short Circuiting Page Table Walk in Multi-GPU Systems via Remote Forwarding. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 456–470.

[33] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, et al. 2022. GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 635–654.

[34] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[35] NVIDIA 2016. *NVIDIA TESLA P100*. NVIDIA. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[36] NVIDIA 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE*. NVIDIA. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[37] NVIDIA 2020. *Pascal MMU Format Changes*. NVIDIA. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf

[38] NVIDIA 2021. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. NVIDIA. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[39] NVIDIA 2024. *NVIDIA Linux Open GPU Kernel Module Source*. NVIDIA. https://github.com/NVIDIA/open-gpu-kernel-modules

[40] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[41] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every Walk's a Hit: Making Page Walks Single-Access Cache Hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 128–141.

[42] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. Colt: Coalesced large-reach tlbs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 258–269.

[43] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 743–758.

[44] Jason Power, Mark D Hill, and David A Wood. 2014. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 568–578.

[45] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2021. Improving GPU Multi-Tenancy with Page Walk Stealing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 626–639.

[46] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2022. Designing Virtual Memory System of MCM GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 404–422.

[47] Nikolay Sakharnykh. 2018. *Everything You Need to Know about Unified Memory*. https://www.nvidia.com/en-us/on-demand/session/gtcsiliconvalley2018-

s8430/

[48] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 180–192.

[49] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 352–363.

[50] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1093–1108.

[51] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. 2023. Memory-Efficient Hashed Page Tables. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1221–1235.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in neural information processing systems* 30 (2017).

[53] Yueqi Wang, Bingyao Li, Aamer Jaleel, Jun Yang, and Xulong Tang. 2024. GRIT: Enhancing Multi-GPU Performance with Fine-Grained Dynamic Page Placement. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE.

[54] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). *ACM SIGMETRICS Performance Evaluation Review* 44, 1 (2016), 337–350.

[55] Shiqing Zhang, Mahmood Naderan-Tahan, Magnus Jahre, and Lieven Eeckhout. 2023. SAC: Sharing-Aware Caching in Multi-Chip GPUs. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3579371.3589078

[56] Xia Zhao, Magnus Jahre, Yuhua Tang, Guangda Zhang, and Lieven Eeckhout. 2023. NUBA: Non-Uniform Bandwidth GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPOLOS), Volume 2*. Association for Computing Machinery, New York, NY, USA, 544–559. https://doi.org/10.1145/3575693.3575745