cuPTW: Leveraging Idle Compute Units for Massively Parallel GPU Page Table Walks

ZIHANG CHEN, The Hong Kong University of Science and Technology (Guangzhou), China
TIANAO GE, The Hong Kong University of Science and Technology (Guangzhou), China
LIEVEN EECKHOUT, Ghent University, Belgium
HONGYUAN LIU, Stevens Institute of Technology, USA
JIAYI HUANG, The Hong Kong University of Science and Technology (Guangzhou), China

Virtual memory has become a cornerstone of modern GPUs, enabling unified address spaces and advanced memory management techniques. However, the performance of address translation has emerged as a critical bottleneck, particularly under irregular workloads with massive memory footprints, where frequent TLB misses and costly page table walks dominate total memory access latency. Prior work has primarily focused on improving TLB effectiveness or optimizing page table walks through batching and coalescing, but these approaches remain limited by the lack of locality and memory bandwidth constraints.

In this work, we propose Compute Unit Page Table Walk (cuPTW), a novel address translation architecture that repurposes idle GPU compute resources to accelerate page table walks. Specifically, we introduce (1) a single-threaded synchronous cuPTW, which offloads translation requests to idle functional units within compute units, and (2) two optimizations that further reduce latency and improve throughput by caching page table walks in local data store memory (cuPTW-SW) and parallelizing them across multiple SIMD lanes (cuPTW-MT). When combined, cuPTW-Full transforms low-parallelism page table walks into a massively parallel computation task. Our evaluation across 15 representative GPU workloads, including deep learning, graph analytics and scientific simulations, demonstrates that cuPTW-Full achieves a performance speedup of 4.43× on average (up to 76.09×) by improving page table walk throughput by 9.92× on average over our baseline GPU architecture. Compared to state-of-the-art GPU address translation proposals, cuPTW-Full achieves a 1.97× to 2.08× average speedup.

CCS Concepts: • **Computer systems organization → Multicore architectures**; **Single instruction, multiple data**.

Additional Key Words and Phrases: Graphics Processing Unit, Virtual Memory, Address Translation, Page Table Walker

ACM Reference Format:
Zihang Chen, Tianao Ge, Lieven Eeckhout, Hongyuan Liu, and Jiayi Huang. 2026. cuPTW: Leveraging Idle Compute Units for Massively Parallel GPU Page Table Walks. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 2, Article 35 (June 2026), 26 pages. https://doi.org/10.1145/3805633

1 Introduction

Virtual memory has become an essential component of modern Graphics Processing Units (GPUs). For seamless data sharing and efficient memory management, GPU vendors have adopted a unified

Authors' Contact Information: Zihang Chen, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, zchen097@connect.hkust-gz.edu.cn; Tianao Ge, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, tge601@connect.hkust-gz.edu.cn; Lieven Eeckhout, Ghent University, Ghent, Belgium, Lieven.Eeckhout@UGent.be; Hongyuan Liu, Stevens Institute of Technology, New Jersey, USA, hliu96@stevens.edu; Jiayi Huang, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, hjy@hkust-gz.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2476-1249/2026/6-ART35
https://doi.org/10.1145/3805633

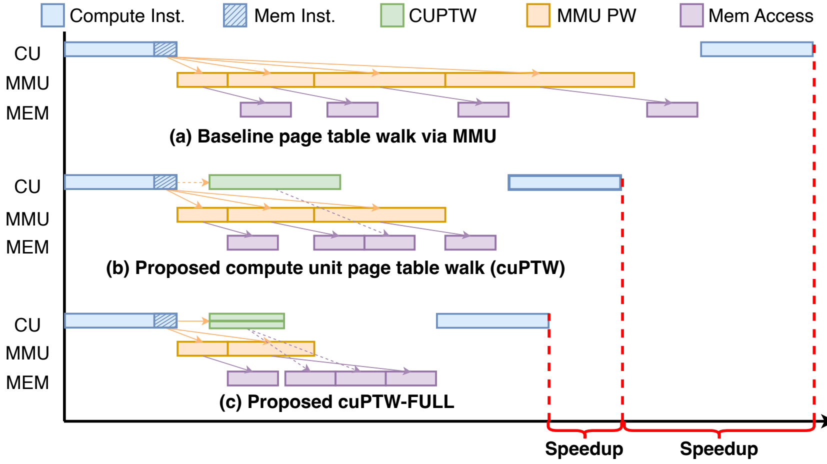

Fig. 1. Comparison of address translation paradigms: (a) Baseline page table walk via MMU; (b) Compute Unit Page Table Walk (cuPTW), which offloads page table walks to the compute units; and (c) cuPTW with caching and multithreading (cuPTW-Full), optimizing the parallel execution of translation requests. The red dashed lines illustrate the speedup achieved by cuPTW and cuPTW-Full.

address space [10, 17, 26, 28, 43, 66], which is fundamentally supported by virtual memory. Virtual memory also facilitates advanced techniques, such as intelligent data placement policies to reduce communication overhead [9, 33, 60]. However, the efficiency of these mechanisms critically depends on the performance of address translation, which has emerged as a major bottleneck in modern GPUs [39, 42, 43, 62].

To accelerate address translation, modern GPUs employ architectural supports such as multi-threaded page walkers for concurrent translation and Translation Lookaside Buffers (TLBs) to cache frequent address mappings [61, 62, 66]. Nevertheless, the ever-growing memory footprint of applications and the increasing prevalence of irregular workloads place tremendous pressure on the address translation subsystem, leading to frequent TLB misses and excessive page table walk requests [20, 21, 47, 50, 62]. Our characterization shows that, in such scenarios, address translation latency can dominate the total memory access time, significantly undermining the immense computational throughput of modern GPUs. This challenge is particularly pronounced in emerging workloads such as large-scale deep learning, graph analytics, and scientific simulations, which feature massive memory footprints and highly irregular memory access patterns [21, 50, 62].

To mitigate the address translation overhead, prior research has primarily explored two directions. The first aims at improving TLB effectiveness through techniques such as page promotion and TLB entry coalescing [10, 37, 55]. However, these approaches are often less effective for irregular workloads with poor spatial locality. The second line of work focuses on optimizing the page table walk process itself, such as batching or coalescing page table walks to exploit locality among page table entries [25, 36, 62]. Unfortunately, the benefits of these techniques are limited by the inherent lack of locality in irregular workloads and the constrained memory bandwidth available to the walker hardware.

To understand the impact of address translation on GPU architecture, we analyze representative GPU applications in terms of translation latency and compute resource utilization. This analysis reveals that the inefficiency of the address translation system forces a large number of execution

threads to stall on long-latency page walks, leading to severe under-utilization of a GPU's computational resources and memory bandwidth. Interestingly, this widespread resource idleness presents a unique opportunity to address the root cause of the translation bottleneck. Our key insight is to repurpose idle compute resources within GPU compute units to perform page walks. By offloading translation requests to functional units and parallelizing the walk process, we substantially increase page walk throughput and overall translation efficiency.

Building on this insight, we propose Compute Unit Page Table Walk (cuPTW), a novel address translation architecture that transforms page table walks from a low-parallelism hardware activity into a massively parallel computation task. The core mechanism of cuPTW involves (1) offloading pending translation requests from the address translation system to compute units, and (2) leveraging idle functional units within these compute units to perform page table walks in parallel. This conversion effectively eliminates the fundamental bottleneck of limited walker parallelism, unlocking the full performance potential of the GPU.

Figure 1 provides an overview of our proposed approach. Compared to the baseline page table walk via the MMU, cuPTW offloads translation requests to the compute units, enabling massively parallel page walks. However, it still faces two challenges. First, cuPTW requires a complete four-level page table walk, which inherently involves multiple memory accesses and results in longer translation latency. Second, each translation wavefront currently executes in a single-threaded manner, leading to low resource utilization. To address these issues, we propose a software-managed page walk cache to accelerate the translation process (cuPTW-SW), as well as a multi-threaded execution model for higher resource utilization (cuPTW-MT), which when combined yields our final proposed design (cuPTW-Full). In summary, this work makes the following contributions:

- We demonstrate how address translation affects memory access latency and overall performance in real GPU applications. We reveal that address translation inefficiency causes underutilization of GPU compute units and we further characterize the translation system to identify the architectural components that dominate GPU translation performance.
- We propose cuPTW, an address translation architecture that utilizes idle compute units to process page table walks that cannot be handled timely by hardware page walkers due to massive TLB misses. We introduce two optimizations based on cuPTW: caching page table walks in local data store memory (cuPTW-SW) and parallelizing them across multiple SIMD lanes (cuPTW-MT). These optimizations further reduce latency and improve throughput in address translation.
- Our evaluation shows that cuPTW, cuPTW-SW, cuPTW-MT and cuPTW-Full achieve a 3.74×, 4.17×, 4.30× and 4.43× geometric mean speedup across 15 applications, respectively. Moreover, we report that cuPTW-Full outperforms the state-of-the-art GPU address translation systems, Marching Page Walkers [36] and SnakeByte [37], by 2.08× and 1.97×, respectively.

2 Background

2.1 GPU Execution Model

Modern GPUs employ the Single Instruction, Multiple Thread (SIMT) execution model to achieve massive computational throughput [4, 51]. At a high level, each GPU kernel consists of multiple work-groups (equivalent to thread blocks in NVIDIA GPUs), each of which comprises work-items (equivalent to threads in NVIDIA GPUs) [4, 51]. Work-items within the same work-group can cooperate through fast on-chip memory and synchronization primitives. At a lower level, work-items are organized into a fundamental execution unit called a wavefront (or warp in NVIDIA terminology), typically consisting of 64 work-items. All work-items in a wavefront share a single program counter and execute the same instruction in lockstep. Consequently, if any work-item in a

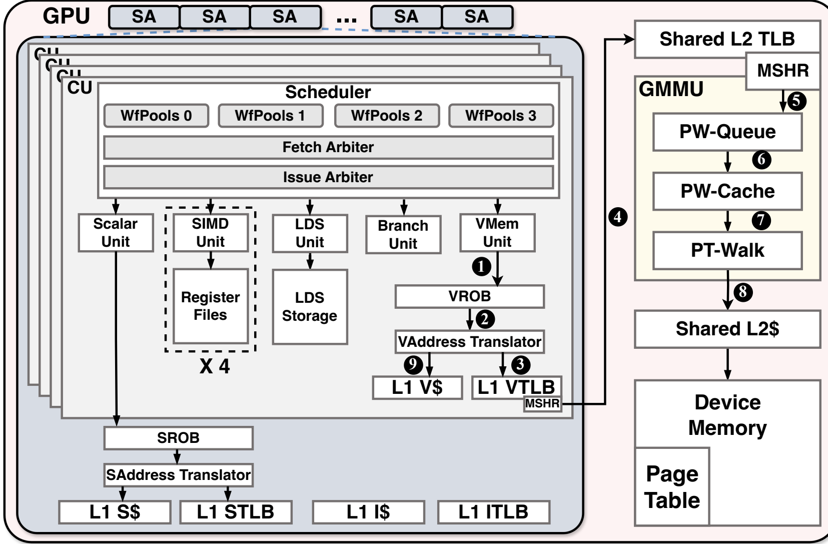

Fig. 2. Overview of the baseline GPU architecture and address translation flow.

wavefront encounters a long-latency operation (e.g., a global memory access), the entire wavefront stalls until the operation completes.

2.2 GPU Compute Unit

To support massive parallelism, modern GPUs integrate a large number of Compute Units (CUs) for instruction execution and data processing [1, 2, 5, 8, 44]. Each CU typically comprises a wavefront scheduler, multiple functional units, and a lightweight on-chip memory hierarchy.

The wavefront scheduler maintains dozens of active wavefronts and can issue multiple instructions per cycle to the available execution units. The major functional units of a CU include:

- **Scalar Unit**: Handles uniform operations shared by all work-items in a wavefront, such as control flow, pointer arithmetic, and the dispatch of common constants, which is accessed at the wavefront level.
- **SIMD (Single Instruction Multiple Data) Units**: A CU typically contains multiple SIMD units (e.g., four 16-lane units) that serve as the primary data-parallel execution engines. Each lane incorporates a single-precision floating-point unit, enabling SIMD execution in lockstep across the assigned wavefronts [14, 46].
- **Vector Memory Unit**: Processes vector-load and vector-store operations between the CU and the global memory hierarchy.
- **Local Data Share (LDS) Unit**: Manages accesses to the LDS, a high-bandwidth on-chip memory that enables efficient data sharing among wavefronts within the same work-group.
- **Branch Unit**: Executes control flow instructions.

Each CU also incorporates a hierarchical on-chip memory subsystem, consisting of register files, the Local Data Share (LDS) [3, 13], and L1 caches. The scalar and SIMD units each have dedicated register files, including a Scalar General-Purpose Register (SGPR) file and a Vector General-Purpose Register (VGPR) file. To facilitate efficient intra-workgroup data sharing, the CU integrates a 32-bank LDS. Additionally, each CU includes a private L1 vector data cache to serve vector memory

Table 1. Configuration details of the baseline GPU.

Module	Configuration
# of CU	1.0 GHz, 128 in total
CU	4 exec. units per CU, 64 threads per wavefront
DRAM	1 TBps, 100 ns latency
L1 V-cache	64 KB, 16-way set associative, 28 cycles, CU private
L1 S-cache	64 KB, 16-way set associative, 28 cycles, shared b/w 4 CUs
L1 I-cache	64 KB, 16-way set associative, 28 cycles, shared b/w 4 CUs
LDS	32 KB, 32 banks, 22 cycles (5 cycles bank conflict penalty), CU private
L2 cache	8 MB, 16-way set associative, 160 cycles, GPU shared
L1 TLB	32 entries, 8 MSHR entries, fully associative, 20-cycle lookup latency, CU private
L2 TLB	2048 entries, 256 MSHR entries, 8-way, 80-cycle lookup latency, GPU shared
Page table walk	16 page table walkers, GPU shared [61, 62, 66]
Page walk cache	32 entries fully associative, 10-cycle lookup latency [16, 59]

accesses, while a shared L1 scalar data cache and L1 instruction cache are typically shared among a small cluster of CUs (e.g., four) to efficiently handle scalar operations and instruction fetches.

2.3 GPU Address Translation

To support memory virtualization, modern GPUs employ a multi-level Translation Lookaside Buffer (TLB) hierarchy and a page-walk subsystem for address translation [56, 58, 59, 61, 62, 66]. In general, the TLB hierarchy comprises a compute-unit–private L1 TLB and a GPU-shared L2 TLB. Figure 2 illustrates the address translation process of the baseline GPU architecture. A memory access request generated by the vector memory unit is first placed into a reorder buffer (❶) and then dispatched to the vector address translator (i.e., VAddress Translator) (❷). In a virtually indexed, physically tagged (VIPT) L1 data cache ([49]), the translator uses the virtual address to index the cache while simultaneously querying the L1 TLB for translation (❸). Upon an L1 TLB miss, the Miss Status Handling Registers (MSHRs) ([35]) are checked to coalesce or filter duplicate translation requests. If no duplicate exists, the request proceeds to the shared L2 TLB (❹). If the L2 TLB also misses, the request is sent to the Graphics Memory Management Unit (GMMU). Due to the limited number of hardware page walkers, incoming requests are queued in the page walk queue (❺). When a page walker becomes available, it first consults the page walk cache [11, 16, 59] to accelerate the page walk by bypassing intermediate levels of the page table (❻). If no hit occurs, the walker iteratively computes the physical addresses of Page Table Entries (PTEs) (❼) and issues memory requests to the shared L2 cache to fetch them (❽). Upon retrieving the final PTE, the page walker completes the process and fills the translations into the TLB hierarchy [19, 52, 58, 59, 61]. After retrieving the final PTE and updating the TLB hierarchy, the translator receives the physical address and finalizes the tag matching in the L1 data cache to complete the memory access (❾).

3 Methodology

3.1 Baseline Configuration

Our evaluation is conducted using MGPUSim [63], configured with the baseline parameters shown in Table 1. We assume a four-level radix-tree–based page table, where the levels are denoted as L4 (root), L3, L2, and L1 (leaf) [56]. Each page table page contains 512 entries, with each page table entry (PTE) occupying 8 bytes, resulting in a 4 KB page size. Work-groups are scheduled onto CUs following a round-robin policy.

Table 2. List of applications categorized by L2 TLB misses per kilo instructions (MPKI).

Abbr.	Application	L1TLB MPKI	L2TLB MPKI	Memory footprint	L2 TLB MPKI
BFS	Breadth First Search	8.5	0.04	80 MB	Low
SC	Simple Convolution on Matrix	7.6	0.4	2048 MB	Low
J2D	2-D Jacobi Solver	95.7	2.2	2048 MB	Low
FWT	Fast Warshall Transform	4.5	2.3	256 MB	Low
J1D	1-D Jacobi Solver	13.6	3.2	2048 MB	Low
RED	Reduction Kernel	11.9	5.9	2048 MB	Low
ST	Stencil Operation on 2-D Matrix	48.2	64.7	512 MB	Medium
PR	Page Rank Algorithm	94.4	95.5	1024 MB	Medium
MT	Matrix Transpose	184.4	196.4	512 MB	Medium
SPMV	Sparse Matrix Vector Multiplication	2,012.8	416.2	360 MB	Medium
SYR2	Rank-2k of a Symmetric Matrix	1,265.2	1,044.9	192 MB	High
GUPS	Multi-thread, Random Access	1,399.9	1,147.1	1025 MB	High
SYRK	Rank of a Symmetric Matrix	864.2	1,324.8	512 MB	High
ATAX	Matrix Transpose and Vector Multiplication	2,225.8	1,890.8	64 MB	High
BICG	Sub Kernel of BiCGStab Linear Solver	2,173.6	2,127.9	64 MB	High

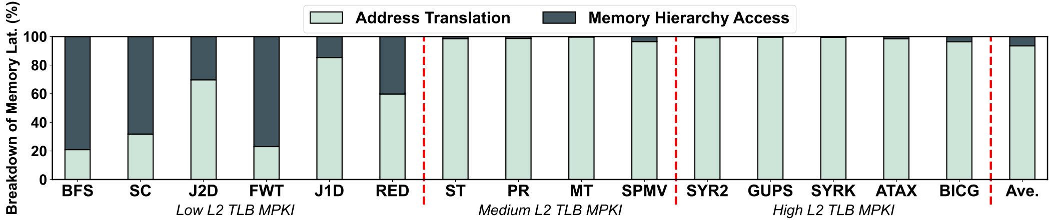

Fig. 3. Breakdown of memory request latency on the baseline GPU.

3.2 Applications

We evaluate 15 benchmarks with diverse page access patterns and memory footprints from AMD APP SDK [8], Hetero-Mark [64], HPCC [45], Pannotia [22], PolyBench [57], and SHOC [23] benchmark suites, as listed in Table 2. The applications cover a wide range of data access patterns and memory intensities. We analyze the page table walk intensity of each application by measuring the misses-per-kilo-instructions (MPKI) at the L2 TLB during address translation. Based on the L2 TLB MPKI, we classify these applications into three categories: Low (MPKI $\leq$ 10), Medium (10 $<$ MPKI $\leq$ 500) and High (MPKI $>$ 500).

4 Motivation and Characterization

This section characterizes how address translation affects overall GPU performance and identifies the primary architectural factors that limit its efficiency. We first analyze the latency contribution of address translation and its impact on compute resource utilization, and then perform a sensitivity study to identify the dominant bottlenecks within the translation system.

4.1 Overall Application Characteristics

Observation 1: A significant portion of the memory access latency causing wavefront stalls is attributed to address translation. To further understand the root cause of such stalls, we analyze the breakdown of average memory access latency for each application on the baseline GPU. The average memory access latency can be decomposed into two parts: (1) the latency incurred

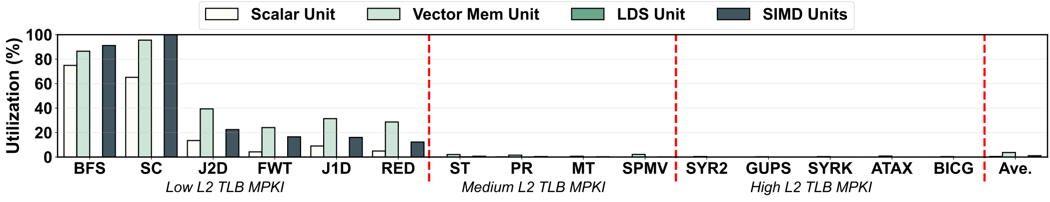

Fig. 4. Overall utilization of functional units for the baseline GPU.

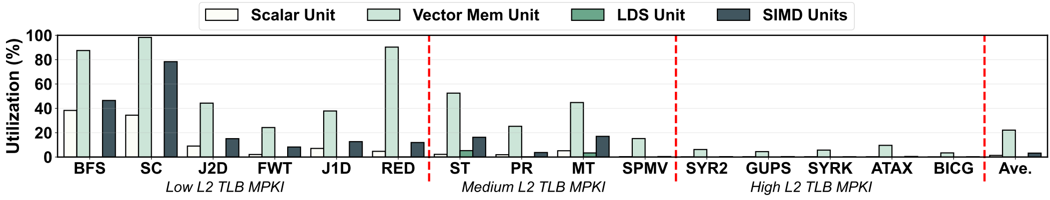

Fig. 5. Overall utilization of functional units for the baseline GPU without address translation overhead.

for address translation, including TLB lookup and page table walks, and (2) the latency required to access data within the memory hierarchy (e.g., cache and DRAM). As shown in Figure 3, our evaluation shows that on average, 92.1% of memory access latency is spent on address translation, whereas only 7.9% is attributed to data access in the memory hierarchy. This result indicates that address translation is the dominant contributor to memory access latency on GPUs, primarily due to frequent TLB misses and page table walks.

Observation 2: Due to the inefficiency of the address translation system, computational and memory resources are under-utilized, significantly reducing overall performance. To illustrate the impact of address translation on GPU resource efficiency, we compare the utilization of different functional units in the compute unit between the baseline GPU and an ideal GPU, where address translation incurs no latency overhead (i.e., no TLB misses or page table walks). The overall utilization of each component is defined as *the ratio of its active cycles to the total compute unit execution cycles (measured from kernel launch to completion).*

As shown in Figure 4, for the baseline GPU, a substantial fraction of hardware resources remains idle during kernel execution, particularly in applications with high MPKI, where frequent TLB misses exacerbate stalls. In the ideal case without address translation overhead, it has a significant increase in the overall utilization of functional units across most applications as shown in Figure 5.

4.2 Address Translation Architectural Characteristics

Observation 3: Among various architectural resources of the address translation system, L2 TLB MSHR capacity and the number of page walkers are the dominant factors determining GPU translation efficiency.

To identify the key factors affecting translation efficiency, we conduct a detailed characterization of the architectural resources in the address translation system. Specifically, we evaluate the performance sensitivity to three key architectural parameters: (1) TLB capacity, (2) TLB MSHR capacity, and (3) the number of page walkers.

Impact of TLB Capacity: A straightforward approach to mitigating address translation overhead is to enlarge the capacity of the L1 and L2 TLBs. Since TLBs cache virtual-to-physical page translations,

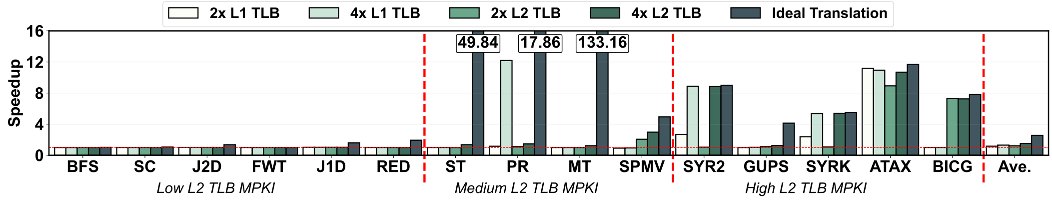

Fig. 6. Speedup varying L1 TLB and L2 TLB capacity versus ideal translation, normalized to the baseline GPU.

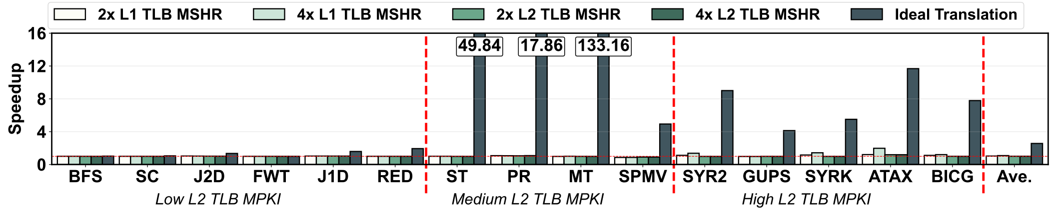

Fig. 7. Speedup when varying the number of L1 and L2 TLB MSHRs vs. ideal translation, normalized to the baseline GPU.

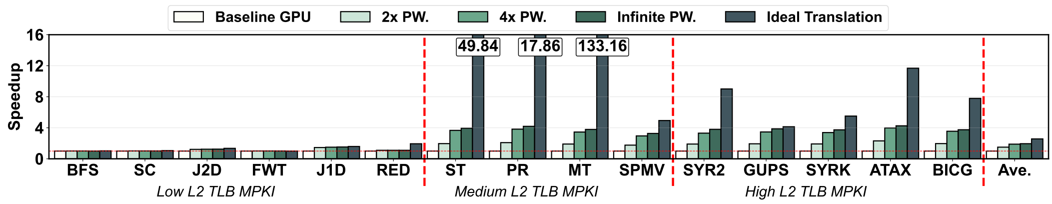

Fig. 8. Speedup when varying the number of page walkers vs. ideal translation, normalized to baseline GPU.

larger TLBs can hold more entries and thus reduce the frequency of costly TLB misses. However, Figure 6 reveals that the benefits of this approach vary widely across workloads. Specifically, ATAX achieves dramatic performance improvements with both 2× and 4× L1 TLB capacity, whereas PR, SYRK, and SYR2 only show improvements when the L1 TLB is quadrupled. In contrast, most benchmarks exhibit negligible or no performance gains with larger L1 TLBs. Even when the L1 TLB capacity is increased by 4×, most applications remain far from the performance upper bound indicated by the Ideal Translation case, where address translation is completely eliminated. These results demonstrate that simply scaling TLB capacity can only close a small portion of the gap and fails to fundamentally address the translation bottleneck. Moreover, larger TLBs incur substantial hardware cost, as highly associative structures consume significant die area and power [12, 18].

Impact of TLB MSHR Capacity: Similarly, we evaluate the impact of TLB MSHR capacity on overall performance, as shown in Figure 7. Larger MSHR capacity allows more concurrent TLB misses to be serviced in parallel, potentially hiding translation latency. Unfortunately, increasing the TLB MSHR capacity shows negligible impact on performance. Even with 2× or 4× the baseline TLB MSHR capacity, none of the benchmarks observe a significant performance improvement.

Impact of Number of Page Walkers: We further evaluate how the number of hardware page walkers impacts overall GPU performance. In contrast to TLB size and L1 TLB MSHR capacity,

which provide workload-sensitive or negligible performance improvements, increasing the number of page walkers significantly benefits most applications (Figure 8). This observation indicates that *the limited concurrency of page table walks constitutes the primary bottleneck in the GPU address translation system.* Notably, even with infinite page walkers, a noticeable performance gap remains compared to the ideal translation scenario. This gap stems from the overhead of traversing the TLB hierarchy and accessing the L2 cache (and occasionally DRAM) during page table walks.

In summary, the low concurrency of page-table walks leads to L2 TLB MSHR saturation, which prevents the L2 TLB from serving subsequent translation requests and eventually stalls address translation. Consequently, both incorporating more page walkers and enlarging L2 TLB MSHR can effectively alleviate translation-induced bottlenecks. However, scaling these translation resources introduces substantial hardware cost in terms of area and power consumption [12, 18, 32, 36].

5 Compute Unit Page Table Walk (cuPTW)

As our characterization in Section 4.2 shows, limited page walk parallelism and L2 TLB MSHR saturation are major bottlenecks in GPU address translation. Meanwhile, a large portion of compute and memory resources in each CU remain idle due to frequent stalls caused by address translation delays as illustrated in Figure 4. These observations motivate our proposal, **C**ompute **U**nit **P**age **T**able **W**alk (**cuPTW**). Briefly, our proposed cuPTW offloads address translation requests to lightweight *translation wavefronts* (detailed in Section 5.1) to perform page table walks using idle functional units (e.g., SIMD and scalar units).

5.1 Overview of Translation Wavefront

A translation wavefront is a special wavefront that performs page table walks by exploiting idle functional units in the compute units. Similar to a normal wavefront following the SIMT execution model, all threads within a translation wavefront collect resources and perform computation in lockstep. In addition, to perform page table walks, a translation wavefront reserves a portion of idle registers to store translation-related metadata (detailed in Section 5.3). Unlike ordinary wavefronts that require an instruction buffer and extensive registers, our translation wavefront requires only a small set of registers, thereby avoiding contention for the limited number of wavefront slots.

In particular, in this section, the number of threads in each translation wavefront is set to one, since our proposed cuPTW design executes page table walks in a single-threaded manner (i.e., only one thread is required to complete the translation procedure). The number of translation wavefronts in each compute unit is set to four to match the number of SIMD units.

5.2 cuPTW Operational Overview

Upon kernel launch, the GPU command processor receives the kernel parameters from the driver and dispatches its work-groups to the compute units in a specific order (e.g., greedy and round-robin [38]). Once all work-groups have been dispatched or the GPU reaches its maximum occupancy, cuPTW probes the available memory resources (i.e., registers and LDS) to initialize a set of translation wavefronts for each compute unit (detailed in Section 5.3). When congestion is detected in the address translation system due to excessive translation requests, some requests are selectively forwarded to the pre-allocated translation wavefronts for performing page table walks (detailed in Section 5.4). The page table walk procedure within a translation wavefront can be decomposed into the following four stages.

Request Setup: When the address translator attempts to forward an address translation request from the TLB hierarchy (❶), it first determines whether an available translation wavefront exists in the same compute unit as shown in Figure 10. If so, it sets up the wavefront's pre-allocated

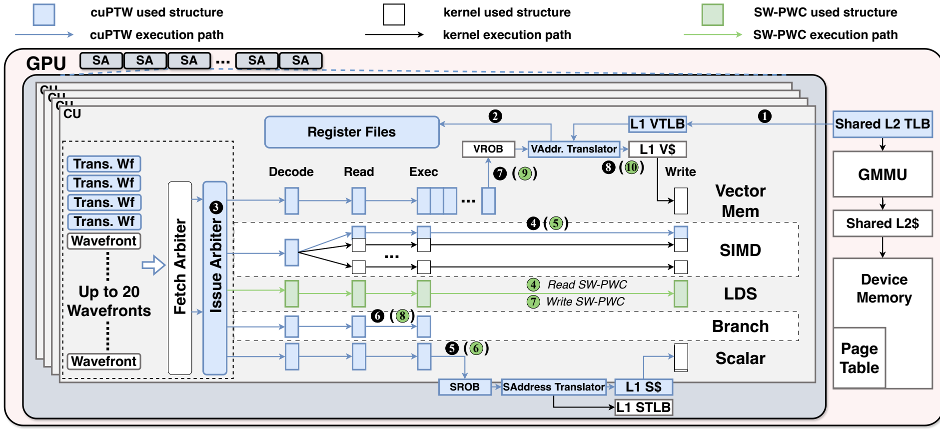

Fig. 9. Overview of our proposed compute unit page table walk and software-managed page walk cache.

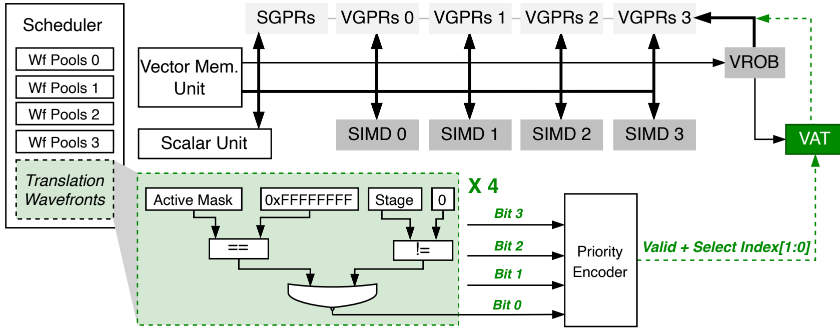

Fig. 10. The Compute Unit including the vector reorder buffer (VROB) and vector address translator (VAT).

registers with the virtual address and address space identifier (ASID) from the incoming request and the value of Page Table Base Register (PTBR), which provides the root address of the page table hierarchy (❷). In addition, it also sets the bits of the 64-bit Vector Condition Code (VCC) register and `active mask` register to one.

After completing the translation request setup, the address translator notifies the wavefront scheduler to initialize the context of the corresponding translation wavefront. The translation wavefront is consequently marked as eligible and its translation stage is set to "Page Offset Calculation" (detailed in Section 5.3). As outlined in Section 4.1, the functional units within the compute unit experience frequent stalls due to inefficiencies in the address translation system. To maximize hardware utilization, if all wavefronts within the wavefront pools are not ready to issue, the wavefront scheduler opportunistically issues translation wavefronts (❸). This mechanism ensures that translation wavefronts only execute when sufficient compute resources are idle, preventing interference with kernel execution. Similar to a normal wavefront, a translation wavefront is considered *eligible* for execution once it has been set up and has no outstanding memory accesses. The issue arbitrator employs a round-robin policy to scan across SIMD units and select eligible translation wavefronts from the pool that are ready to be issued for page offset calculation.

Page Offset Calculation: In this stage, a wavefront is selected for issuance when it is marked as eligible and its internal translation stage is "Page Offset Calculation". This stage is responsible for

calculating the offset of the target page table entry (PTE) that contains the page frame number for the next level of the page table hierarchy.

Once selected, the wavefront is issued to the corresponding SIMD unit to compute the offset for the PTE address (❹). The computation begins by loading the virtual address and the current page walk level from its dedicated registers. Next, using Equation 1, the SIMD unit shifts and masks the virtual address to extract the level-specific virtual page number (VPN) based on the current page walk level. The computed page offset is then stored in the dedicated register. After that, if the level reaches 4, the SIMD unit sets the bits of the VCC register to zero. Finally, the SIMD unit updates the translation stage to the "Sent to Memory" stage, indicating that the current PTE offset computation is complete. Meanwhile, the calculated offset can be combined with the base address to generate the physical address for accessing the next-level page table.

$$\text{Page Offset} = (\text{Address} \gg (12 + 9 \times (4 - \text{Level}))) \ \& \ 0\text{x}1\text{FF} \tag{1}$$

Sent to Memory: Similar to the "Page Offset Calculation" stage, when no kernel wavefronts are available in the wavefront pools, the wavefront scheduler and issue arbitrator select an eligible translation wavefront whose translation stage is "Sent to Memory". The selected wavefront is then issued to the scalar unit (❺). Subsequently, the scalar unit reads the dedicated registers from the register file to extract the page offset computed in the previous stage. It then generates the physical address of the next-level PTE by combining the computed page offset with the base address of the next page level. Finally, the scalar unit issues a memory access request for this *physical* memory address to the scalar reorder buffer (SROB). To distinguish these internal memory requests, we augment each memory packet with a 1-bit cuPTW flag. If a request originates from a translation wavefront, the scalar unit asserts this flag. Upon packet arrival, the scalar address translator checks this bit; if asserted, the request is identified as a PTW access, allowing it to bypass the L1 scalar TLB and directly access the scalar cache using the pre-calculated physical address. Otherwise, the request follows the conventional translation data path.

In particular, cuPTW maintains a unified caching policy in the scalar cache for both data and page table walk requests. Therefore, page table walk requests follow the same lookup and miss-handling mechanism as regular data accesses. Likewise, fetched page table entries retrieved from lower-level memory hierarchies are filled in the scalar cache. In our evaluation, caching PTEs in the L1 scalar cache has a negligible impact on normal data access hit rates (detailed in Section 7.2). Once a PTE is retrieved from the L1 scalar cache, the contained page frame number (PFN) is written into the dedicated register of the source translation wavefront. The wavefront stage is subsequently updated to "Termination Check".

Termination Check: At this stage, the wavefront is dispatched to the branch unit, which performs a bitwise AND operation between the `active mask` register and the VCC register to determine whether the translation is complete (❻). If the result is zero, the stage transitions to "Translation Done"; otherwise, it resets to "Page Offset Calculation" to compute the offset for the next-level PTE.

Translation Done: Once in the "Translation Done" stage, the wavefront is issued to the vector memory unit, which extracts the PFN from dedicated registers and forwards the physical address to the address translator (❼). The translator then resolves the pending cache access requests (❽). Meanwhile, the translation result is cached within the L1 and L2 TLBs to update the translation hierarchy.

5.3 Translation Wavefront Organization

To maximize hardware utilization, modern GPUs employ a zero-overhead context-switching mechanism that instantly issues ready wavefronts to idle functional units whenever executing wavefronts

Table 3. Remaining GPU Resources per CU across Applications.

Abbr.	LDS (Ratio %)	SGPR (Ratio %)	VGPR per SIMD (Ratio %)
BFS	65536 (100.0%)	1920 (60.0%)	136.0 (0.8%)
SC	65536 (100.0%)	1920 (60.0%)	96.0 (0.6%)
J2D	65536 (100.0%)	2560 (80.0%)	96.0 (0.6%)
FWT	65536 (100.0%)	2560 (80.0%)	136.0 (0.8%)
J1D	65536 (100.0%)	2560 (80.0%)	136.0 (0.8%)
RED	65024 (99.2%)	3136 (98.0%)	250.0 (1.5%)
ST	7168 (10.9%)	2816 (88.0%)	16.0 (0.1%)
PR	55296 (84.4%)	1920 (60.0%)	96.0 (0.6%)
MT	24576 (37.5%)	1920 (60.0%)	16.0 (0.1%)
SPMV	65536 (100.0%)	1280 (40.0%)	136.0 (0.8%)
SYR2	65536 (100.0%)	1920 (60.0%)	96.0 (0.6%)
GUPS	65536 (100.0%)	2944 (92.0%)	240.0 (1.5%)
SYRK	65536 (100.0%)	1920 (60.0%)	136.0 (0.8%)
ATAX	65536 (100.0%)	3192 (99.8%)	254.5 (1.6%)
BICG	65536 (100.0%)	3192 (99.8%)	254.5 (1.6%)
Geometric Mean	52343.6 (79.9%)	2310.2 (72.2%)	109.6 (0.7%)

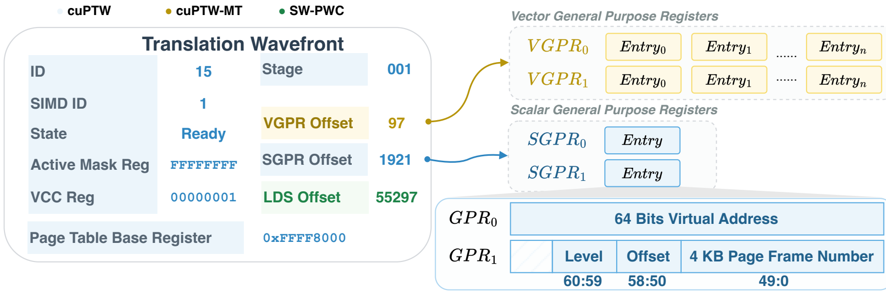

Fig. 11. Overview of the translation wavefront.

stall on long-latency memory operations. However, supporting such rapid switching requires dedicating a significant portion of on-chip memory to storing wavefront state including fetched instructions and a large amount of allocated register files, which ultimately limits the number of wavefront slots in each SIMD unit. In contrast, our proposed translation wavefronts are fundamentally lightweight. They do not require instruction fetching or decoding and only use a minimal number of registers, significantly reducing per-wavefront state storage. As a result, they impose negligible hardware overhead and avoid contention for the critical compute wavefront slots reserved for application kernels [15, 54].

To seamlessly integrate translation wavefronts into the existing context-switching mechanism, each translation wavefront is associated with a minimal context structure. Similar to the kernel wavefront structure [1], this context includes a unique wavefront ID, current state and translation stage, pointers to its allocated SGPRs, VGPRs and LDS memory, the assigned SIMD ID, active mask register and VCC register, and page table base register (PTBR), as illustrated in Figure 11.

As shown in Table 3, GPU workloads typically leave a significant portion of on-chip resources underutilized. In particular, most benchmarks leave a large fraction of SGPRs unallocated (73.4% on average). These available on-chip memory resources can be repurposed to store page table walk information for translation wavefronts. In our design, idle SGPRs are exploited to store page

table walk metadata for *translation wavefronts*. Specifically, each thread in a translation wavefront is allocated two idle SGPRs to store page table walk metadata. One for address data and one for intermediate translation state. Since cuPTW employs single-threaded translation wavefronts, each wavefront requires only two SGPRs in total, as illustrated in Figure 11. The first register (GPR_0) stores the target 64-bit virtual address, while the second register (GPR_1) holds the intermediate page table walk state, including the current level, computed page offset, and fetched PFN.

To avoid contention for critical on-chip resources (i.e., registers and LDS) required by work-groups and thus prevent degrading GPU occupancy, idle resources are allocated to translation wavefronts only after all work-groups have been dispatched or the GPU reaches maximum occupancy (i.e., no additional work-group can be dispatched due to resource constraints).

5.4 Translation Request Forwarding

To efficiently utilize compute units for page table walks, cuPTW adopts a selective forwarding mechanism at the L2 TLB. Specifically, **when the L2 TLB's MSHRs become saturated, newly missed translation requests (i.e., those that cannot be serviced due to MSHR unavailability) are offloaded to compute units for page table walks.** We choose L2 TLB forwarding as it strikes a balance between responsiveness and accuracy. Forwarding from earlier stages (e.g., the address translator or L1 TLB) may prematurely offload requests that could still be resolved by deeper TLB levels, resulting in redundant page table walks and unnecessary contention for compute resources. In addition, cuPTW avoids performing forwarding at the page walk queue because this approach still suffers from back-pressure imposed by the limited L2 TLB MSHRs. Specifically, when the L2 TLB MSHRs are full, the page walk queue can receive and forward new translation requests only after the corresponding MSHR entries have been released. Consequently, even if missed translation requests are offloaded from the queue to compute units, the L2 TLB remains unable to service new incoming lookups until these MSHRs are freed. In contrast, performing forwarding directly at the L2 TLB eliminates both redundant offloads and the MSHR-induced stall, allowing the L2 TLB to continuously service subsequent translation requests.

6 Optimizing cuPTW for Latency and Throughput

Our proposed cuPTW effectively improves the throughput of page table walks and mitigates congestion in the address translation subsystem, thereby enhancing the overall GPU performance. However, two key challenges remain. (1) A complete 4-level page table walk inherently involves multiple memory accesses, which increases translation latency. (2) Each translation wavefront executes in a single-threaded manner, resulting in limited resource utilization.

6.1 Compute Unit Page Walk Cache

As discussed in Section 5.3, GPU workloads typically do not fully utilize the LDS memory available within compute units. We therefore repurpose this underutilized on-chip memory resource to accelerate page table walk operations. Specifically, we configure idle LDS memory as a software-managed page walk cache (referred to as **SW-PWC**), thereby reducing the number of memory access requests required by cuPTW.

Challenges: Due to the fundamental architectural differences between the fully associative organization used in traditional GPU MMU page walk caches and the banked memory structure of LDS memory in compute units, we cannot directly employ LDS memory to emulate a conventional hardware page walk cache. The primary difficulty is that the fully associative organization enables the hardware page walk cache to rapidly scan all cache blocks to identify and match the page table

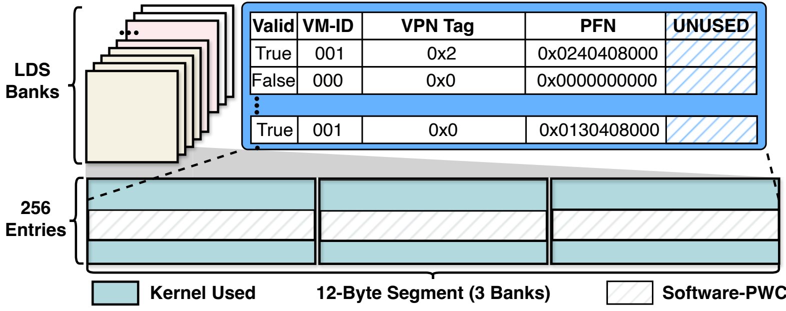

Fig. 12. Overview of software-managed page walk cache.

entry closest to the leaf node, while banked LDS memory cannot support parallel and content-addressable lookup. To address this, we design a lightweight software-managed PWC layout that minimizes bank conflicts while maintaining lookup efficiency.

Software-managed PWC Organization: The baseline GPU architecture features a compute unit containing 32 memory banks of LDS, each comprising 256 entries of 4 bytes. To support parallel matching of three-level page table entries (except L1 PTEs) without incurring bank conflicts, we partition the 32 banks into three dedicated groups, with each group responsible for matching page table entries at one specific level. As shown in Figure 12, within each group, we further organize the entries in each row into multiple smaller 12-byte segments to store both tags and data payloads. To reduce design complexity, we employ a direct-mapped structure for the software-managed page walk cache. In this scheme, the tag consists of a 1-bit `Valid` field, a 2-bit `VM-ID` serving as the ASID and a `VPN` tag. For a 48-bit virtual address space with 4 KB pages, the length of the VPN tag can be derived using Equation 2. The data payload section stores a 40-bit PFN.

$$\text{VPN Tag Length} = 9 \times (4 - \text{Level} + 1) - \log_2(\text{Number of Blocks}) \tag{2}$$

Lookup: The software page walk cache employs a direct-mapped lookup procedure. Assuming that we assign 24 banks with 8 entries each for L3 PTEs, the total number of blocks dedicated to L3 becomes $24 \times 8/3 = 64$ blocks. Consequently, the tag length is calculated as $9 \times (4 - 3 + 1) - 6 = 12$ bits. During lookup, the 6 least significant bits of the VPN index the target cache block (i.e., a 12-byte segment). The software page walk cache then verifies the Valid bit, VM-ID, and VPN tag against the page walk request. On a hit, the corresponding PFN is retrieved from the cache block.

Integration with the pipeline: During the execution of cuPTW, the SW-PWC implemented in LDS memory serves the same role as a hardware page walk cache. When a translation wavefront initiates a page-table walk, the scheduler issues the translation wavefront to the LDS unit. The LDS unit then reads the target virtual address stored in GPR_0 and performs a lookup in the software-managed cache resident in LDS memory (❹). If a hit occurs, the LDS unit writes the retrieved PFN to the designated field in GPR_1. Otherwise, the physical address of the root page table is written to GPR_1 to initiate a full page table walk. The translation wavefront then proceeds through the *Page Offset Calculation* (❺) and *Sent to Memory* (❻) stages. Next, the wavefront is dispatched back to the LDS unit to update the SW-PWC (❼). Specifically, the previous level PFN is written into the SW-PWC. Once the page walk cache update is complete, the wavefront proceeds to the *Page Offset Calculation* stage to compute the offset of the next-level PTE again.

6.2 Multi-threaded Translation Wavefront

In the proposed single-threaded cuPTW, translation throughput is constrained by limited thread-level parallelism, leading to severe underutilization of computational resources within each compute unit. Each SIMD unit in our baseline GPU contains 16 lanes, yet single-threaded cuPTW utilizes only one, achieving a mere 6.25% of its theoretical utilization. This underutilization fundamentally restricts the overall throughput of the address translation process. To better exploit the available computational resources, we propose a multi-threaded design, referred to as **cuPTW-MT**.

While multi-threading appears straightforward, it introduces two key challenges. First, translation threads may progress at different rates due to variable memory access latencies and differing page table levels (arising from the software-managed page walk cache), potentially causing control-flow divergence. Second, the baseline cuPTW stores translation-related metadata in SGPRs, which cannot be accessed concurrently by multiple SIMD lanes, thus preventing efficient parallel execution.

To address these challenges, cuPTW-MT organizes multiple translation threads into a lockstep execution wavefront, ensuring that all threads operate at the same translation stage. Each translation wavefront maintains a 2-bit stage flag shared across all threads, along with an active mask indicating which threads are currently participating in translation. Each bit in the mask corresponds to a thread. Upon receiving translation requests, the address translator sequentially initializes the threads within the translation wavefront and stores the corresponding translation metadata into the VGPR entries associated with the initialized threads. Like single-threaded cuPTW, $VGPR_0$ stores the 64-bit virtual address for each active thread, while $VGPR_1$ holds their corresponding intermediate page table walk state.

Once all threads within the wavefront have been initialized, the wavefront is marked as eligible for execution. To avoid long stalls or potential deadlocks caused by incomplete wavefronts waiting for additional translation requests, we introduce a lightweight timeout mechanism. If a translation wavefront remains partially filled beyond a predefined threshold (e.g., 128 cycles), it is marked as eligible and proceeds with the activated translation threads. This mechanism ensures forward progress while maintaining hardware simplicity.

When a translation wavefront is dispatched to a SIMD unit, the unit first reads its active mask to identify active threads and performs page offset calculations in the corresponding lanes in parallel. After all offsets are computed, the wavefront's translation stage is updated to "Sent to Memory", and the scheduler dispatches it to the scalar unit for PTE fetching. The scalar unit then reads the computed offsets from the $VGPR_1$ entries sequentially with each entry corresponding to one active thread, and issues the corresponding memory access requests. Notably, multiple 8-byte PTE requests often reside within the same cache line, especially for upper-level page-table entries [62]. These memory requests are naturally coalesced by the L1 scalar cache (detailed in Section 7.2).

7 Evaluation

In this section, we evaluate our proposed techniques using MGPUsim. The detailed simulation configurations and the applications are the same as in our characterization studies (listed in Table 1 and Table 2). We measure a kernel's end-to-end execution time for the different GPU page table walk designs, and we compute speedup normalized to our baseline GPU. Average performance is calculated as the geometric mean speedup across all benchmarks. Specifically, we evaluate the following three configurations:

- **Baseline**: A conventional GPU that performs page table walks as described in Section 2.3.
- **MPW**: Marching Page Walks [36], a state-of-the-art design that enhances page walk parallelism by allowing multiple in-flight walks across different levels of the page table.

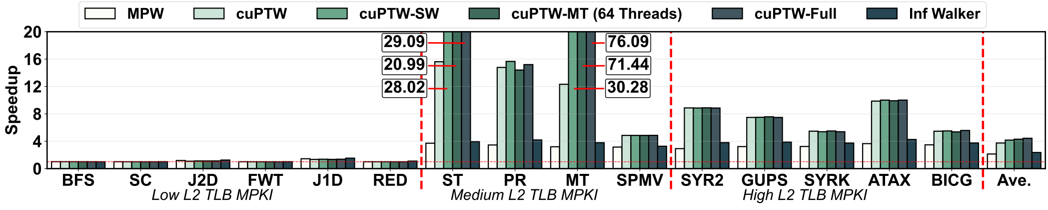

Fig. 13. Speedup for MPW, the cuPTW variants and infinite page walkers, normalized to baseline GPU.

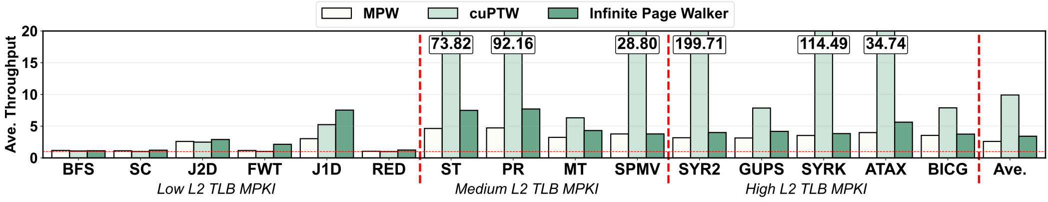

Fig. 14. Average address translation throughput normalized to the baseline GPU.

- **cuPTW**: We consider four variants of our proposed design: (1) our reference cuPTW proposal, (2) enhanced with a software-managed page walk cache (cuPTW-SW), (3) batching translation requests (cuPTW-MT), and (4) cuPTW-Full which combines cuPTW-SW and cuPTW-MT.

7.1 Overall Performance

Figure 13 reports speedup for all evaluated configurations. In general, all of our proposed techniques significantly improve overall performance compared to the baseline GPU and MPW. On average, cuPTW achieves a 3.74× and 1.75× speedup over the baseline and MPW, respectively. cuPTW-SW further reduces the average page-table-walk latency, leading to 1.11× overall improvement over cuPTW. Furthermore, cuPTW-MT improves performance by exploiting higher thread-level parallelism, increasing the average speedup to 1.15× over cuPTW. Compared to MPW, cuPTW-Full improves performance by approximately 2.08× on average. Applications with medium and high L2 TLB MPKI (e.g., ST, MT, and SPMV) benefit the most from our design. This is because cuPTW effectively exploits the available compute units to improve the page walk throughput, which is the primary bottleneck for these workloads. Meanwhile, applications with low L2 TLB MPKI also show a 1.08× average improvement, as cuPTW-Full primarily mitigates translation delays.

For reference, we also include an idealized configuration: an infinite page-walker configuration (**InfPW**) that provides unlimited page-walk parallelism. As shown in Figure 13, compared to the configuration with infinite page walkers, our proposed cuPTW design achieves even higher performance by leveraging the idle bandwidth of L1 caches and the LDS to alleviate contention on L2 caches and the hardware page walk cache.

7.2 Optimization Analysis

Impact of Compute Unit Page Table Walk: To further understand the performance difference among MPW, cuPTW, and InfPW, Figure 14 presents the normalized address translation throughput. For *High-MPKI* applications, our proposed cuPTW substantially improves the address translation throughput. On average, it achieves 9.92×, 3.81×, and 2.90× higher translation throughput compared to the baseline GPU, MPW and InfPW, respectively. For *Low-* and *Medium-MPKI* applications, the

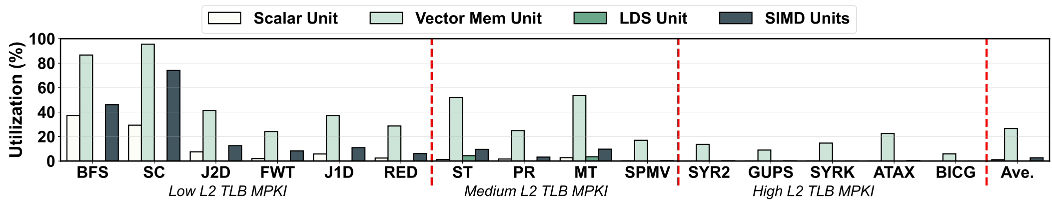

Fig. 15. Overall utilization of functional units for kernel execution with cuPTW.

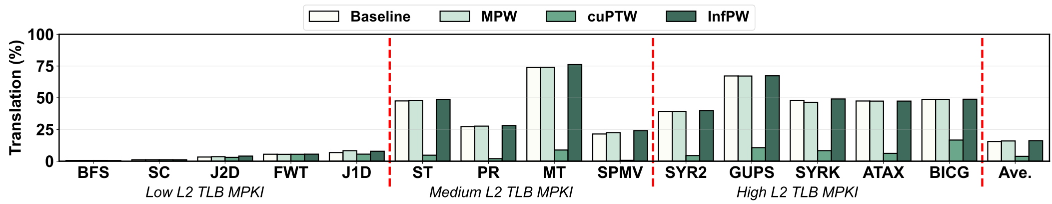

Fig. 16. Fraction of translation accesses in L2 across GPU benchmarks with different methods.

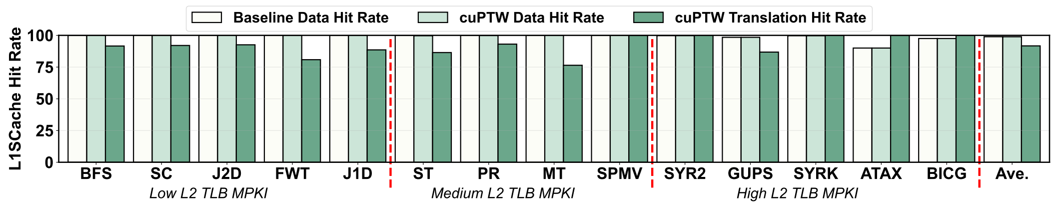

Fig. 17. L1 scalar cache hit rate across GPU benchmarks with baseline GPU and cuPTW.

average translation throughput of cuPTW does not significantly outperform, and can even fall below, that of the MPW and InfPW configurations. However, the overall performance improvement does not correlate directly with its average translation throughput. For most *Low-MPKI* workloads, due to the low translation pressure and computational resource contention, our proposed cuPTW achieves comparable performance to MPW and InfPW. For *Medium-MPKI* applications, some compute units initiate translation requests later than others, leading to long queuing latency in the TLB and MMU, causing significant load imbalance across compute units. We quantify this imbalance using the standardized standard deviation of per-CU executed instruction counts across all benchmarks, which are 0.27, 0.25, 0.22, and 0.09 for the baseline, MPW, infPW and cuPTW on average, respectively. Hence, although cuPTW does not achieves higher average translation throughput than MPW or InfPW, it effectively mitigates this imbalance and consequently delivers significant performance gains.

In addition, we assess the overall utilization of functional units throughout kernel execution under the cuPTW design to understand its impact on compute resource efficiency, as shown in Figure 15. Compared to the baseline GPU (as shown in Figure 4), the proposed cuPTW design significantly increases the page table walk throughput and mitigates address translation overheads. As a result, it effectively improves the utilization of the vector memory units and slightly enhances the efficiency of computational resources, including both the scalar and SIMD units.

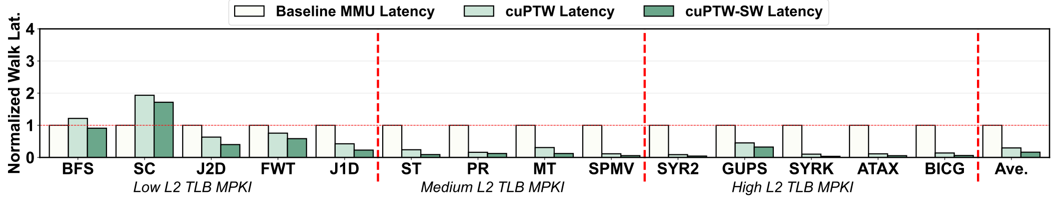

Fig. 18. Normalized page walk latency across GPU benchmarks with different methods, normalized to the baseline GPU MMU.

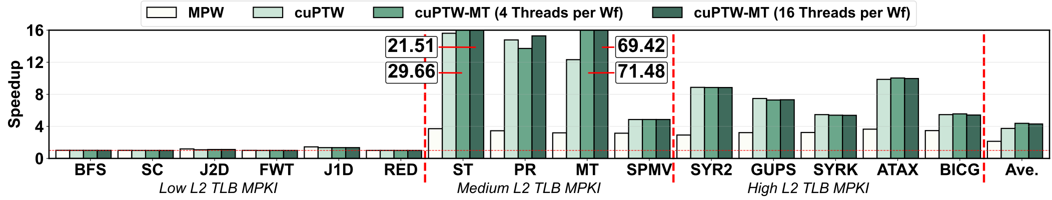

Fig. 19. Speedup of GPU benchmarks with MPW, cuPTW, and cuPTW-MT (4 and 16 threads per wavefront), normalized to the baseline GPU.

Furthermore, we analyze the impact of cuPTW on the memory subsystem. As shown in Figure 16, the fraction of page table walk-related memory accesses to L2 remains high in both the MPW and InfPW designs, since they only enhance the page table walk parallelism without mitigating cache contention. In contrast, by offloading page table walks to compute units, cuPTW leverages the underutilized L1 bandwidth, effectively reducing L2 traffic and easing the overall pressure on the memory subsystem. Moreover, we also demonstrate the L1 scalar cache hit rate for both data access and page table walk, as shown in Figure 17. In general, cuPTW achieves a high translation hit rate, while the data hit rate remains unaffected, indicating that performing page table walks within compute units does not interfere with normal data accesses.

Impact of Software-Managed Page Walk Cache: To understand how SW-PWC improves translation efficiency, we analyze the latency of page table walks performed by compute units under different designs. Figure 18 compares the average compute unit page walk latency of cuPTW and cuPTW-SW against the page walk latency via the baseline GPU MMU. The results show that the latency of page walks performed by compute units is 3.38× lower than that of the hardware MMU, with the average latency still exceeding 500 cycles. However, by exploiting available LDS to store page table entries, SW-PWC accelerates the cuPTW page walks by an average of 1.81×.

Impact of Multi-Threaded Compute Unit Page Table Walk: We use different numbers of threads (1, 4, 16) in the translation wavefront to demonstrate the throughput improvement from multi-threading. As shown in Figure 19, cuPTW-MT (16 Threads) achieves a speedup of up to 5.63× and an average improvement of 1.15× across all workloads compared to cuPTW. For most workloads, cuPTW-MT with 4 and 16 threads achieve similar speedups, indicating that increasing the number of threads beyond 4 provides limited additional performance gains.

We further analyze the average number of active threads in the translation wavefronts with 4 and 16 threads per wavefront in cuPTW-MT. We observe that increasing the number of threads from 4 to 16 does not significantly improve translation parallelism in most workloads, except for MT and GUPS. This is because the high throughput of compute units in cuPTW-MT effectively alleviates

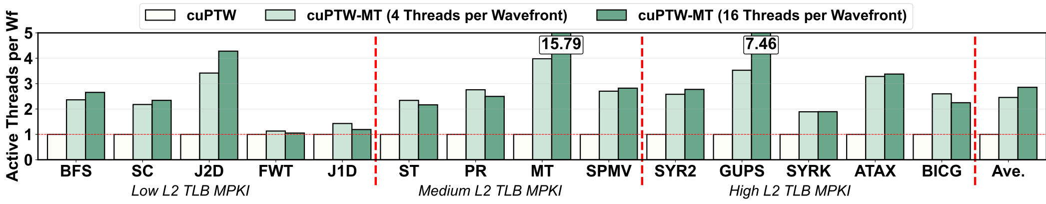

Fig. 20. Number of active threads for cuPTW-MT (4 and 16 threads per wavefront) normalized to cuPTW.

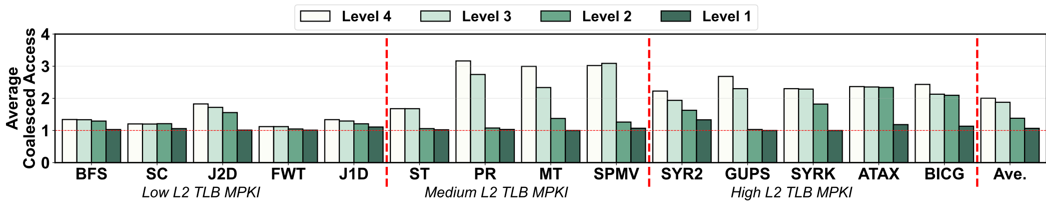

Fig. 21. Average number of coalesced memory accesses in L1 for cuPTW-MT (4 threads per wavefront).

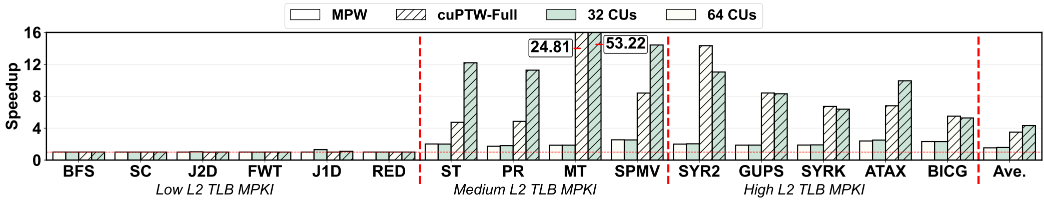

Fig. 22. Speedup for MPW and cuPTW-Full as a function of the number of compute units, normalized to the baseline GPU.

the pressure on the address translation system, leaving few outstanding walks to exploit additional translation parallelism. As a result, there is limited opportunity to exploit additional parallelism when the number of threads is increased beyond 4. We also examine how cuPTW-MT interacts with the L1 scalar cache during page table walks. Figure 21 indicates the average number of coalesced memory requests observed in the L1 scalar cache across different page-table levels. We observe that higher-level page table accesses (e.g., L4 and L3 levels) typically exhibit strong coalescing effects, as translation threads often fetch contiguous upper-level entries. Since our cuPTW repurposes the scalar unit's data access path to perform page table walks, the L1 scalar cache automatically attempts to coalesce memory requests issued from the same translation wavefront. This behavior allows multiple threads within a wavefront to access identical cache lines through a single memory transaction, thereby improving data locality and reducing cache port pressure.

7.3 Sensitivity Analyses

Impact of Number of Compute Units: Since cuPTW leverages compute units (CUs) to execute page table walks, the number of CUs has a significant impact on performance. Figure 22 reports normalized performance as the number of CUs increases from 32 to 64, relative to a GPU with the same CU count but without cuPTW support. For most applications, we observe that performance improves proportionally with the number of CUs, indicating the strong scalability of our proposed

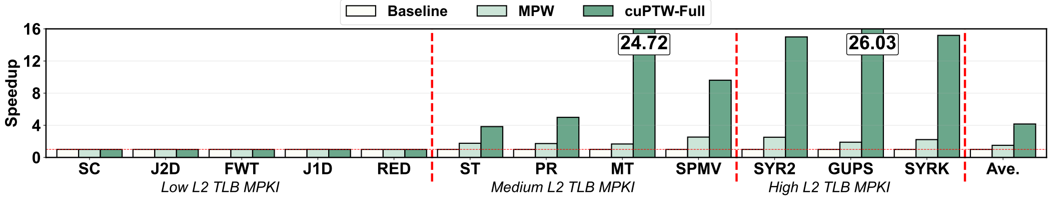

Fig. 23. Speedup for MPW and cuPTW-Full in a Multi-Chiplet Module GPU setup.

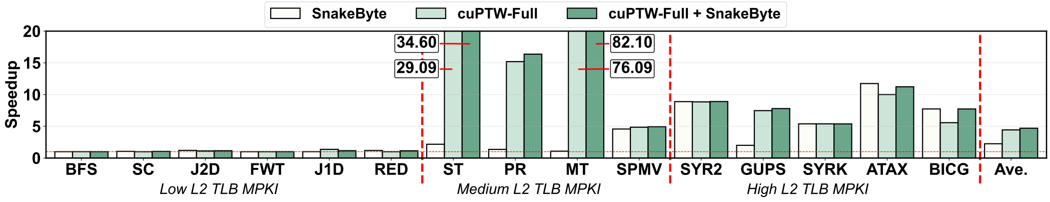

Fig. 24. Speedup for SnakeByte, cuPTW-Full, and the combination of cuPTW-Full and SnakeByte, normalized to a baseline GPU with 4 KB pages.

techniques. For a few applications (e.g., SYRK and SYR2), the observed speedup slightly decreases as the CU count grows. In these cases, the larger number of CUs issues more memory requests, which increases contention and pressure on the memory hierarchy. Although the translation throughput itself continues to rise, the overall speedup is constrained by memory bandwidth saturation.

Performance Improvements on Multi-Chiplet Module GPUs: Multi-Chiplet Module GPUs (MCM-GPUs) are becoming increasingly popular due to their scalability and cost efficiency [9, 25, 60, 69]. However, in MCM-GPUs, a page table walk may need to access page table entries located in a remote memory partition via a low-bandwidth interconnect, which increases page table walk latency and amplifies pressure on the address translation subsystem compared to monolithic GPUs [25, 60]. To understand the impact of our proposed techniques on MCM-GPUs, we conducted a series of experiments following the MGvm configuration [60], employing a four-chiplet MCM-GPU architecture. We also integrate the LASP [33] mechanism to reduce remote data accesses. Figure 23 shows the performance comparison of the baseline MCM-GPU, MPW, and our cuPTW-Full design. On average, cuPTW-Full delivers 4.16× and 2.76× higher performance than the baseline and MPW, respectively, demonstrating the effectiveness of cuPTW even in a distributed memory system.

8 Discussion

Comparison with Superpage-Based Schemes: To further demonstrate the efficacy of cuPTW, we compare it against state-of-the-art superpage-based translation mechanisms (e.g., SnakeByte [37]), which incorporates a contiguity-aware demand paging policy [7, 26] that allocates 16 consecutive pages (64 KB) when a new page is allocated. Following the experimental methodology of prior studies [30, 37, 61], we exclude the data migration overhead of demand paging. As shown in Figure 24, SnakeByte enhances TLB capacity by recursively coalescing eight 4 KB base pages into larger superpages up to 8 GB, effectively reducing L2 TLB miss rate by 2.85× and achieving a 2.25× average speedup. However, for applications with non-contiguous access patterns (e.g., MT and GUPS), SnakeByte is limited to a 32 KB page granularity as it can only coalesce eight contiguous 4 KB base pages. This limitation results in high L2 TLB miss pressure, which significantly degrades performance. By exploiting massive idle compute units, our proposed cuPTW effectively eliminates

Table 4. Synthesis Results for the Address Translator (ASAP 7nm Library).

Design	**Area (mm^2)**	**Delay @ 1 GHz (ps)**	**Delay @ 2 GHz (ps)**
Baseline Design	0.27418	980.67	475.92
Design with Bypass Logic	0.27439	987.24	482.09
Overhead	**+0.00021 (+0.077%)**	**+6.57**	**+6.17**

translation bottlenecks and outperforms SnakeByte by 1.97×. Furthermore, we evaluated cuPTW on top of SnakeByte, resulting in 2.09× speedup over SnakeByte.

Power/Energy Consumption: To estimate GPU power and energy consumption, we integrated AccelWattch [31] into MGPUSim [63]. Our evaluation indicates that cuPTW increases average GPU power by 11.5% compared to the baseline GPU. Notably, the power overhead incurred by additional register accesses and integer computations remains negligible. Instead, the power increment is predominantly attributed to the alleviation of translation bottlenecks, which facilitates more parallel cache/DRAM accesses and improves overall computational efficiency. Due to the reduction in execution time, cuPTW effectively reduces the overall GPU energy consumption by 3.68× on average, and up to 53.02×.

Hardware Overhead: To support cuPTW, a translation wavefront must maintain a light-weight context as described in Section 5.3. Specifically, consistent with the kernel wavefront structure [1], a translation wavefront requires an additional 4 bits (wavefront ID) + 2 bits (SIMD ID) + 3 bits (wavefront state) + 3 bits (translation stage) + 8 bytes (active mask register) + 8 bytes (VCC register) + 6 bits (VGPR Offset) + 6 bits (SGPR Offset) + 8 bits (LDS Offset) + 8 bytes (PTBR) = 224 bits. Assuming a GPU configuration of 128 compute units, each supporting 4 concurrent translation wavefronts, the aggregate overhead for the entire GPU amounts to 14,336 bytes. To minimize hardware overhead, the communication from the L2 TLB to the Vector Address Translator (VAT) leverages existing data paths. Similarly, the write-back process from the VAT to the register file reuses the data paths of the VROB. Consequently, the hardware overhead for translation request forwarding is limited to a 3-bit signal line from the scheduler to the VAT as shown in Figure 10. Given that our baseline GPU is similar to the AMD MI100 GPU (TSMC 7nm process), we implement the incurred hardware in RTL and use Synopsys Design Compiler with the ASAP 7nm library [65] to estimate additional chip area. The results show that the area overhead introduced by translation wavefronts is approximately 0.01346 mm^2, accounting for only 0.0018% of the total GPU die area (750 mm^2). Furthermore, we introduce bypass logic that allows cuPTW memory requests to bypass the scalar TLB lookup and directly access the L1 scalar cache with a pre-calculated physical address. As shown in Table 4, this bypass logic incurs an area overhead of only 0.000211 mm^2. Beyond area considerations, we also perform a rigorous timing evaluation of the introduced bypass logic within the scalar data path using Synopsys Design Compiler. Synthesis results indicate that under a 1 GHz clock constraint, the bypass logic introduces a combinatorial delay of only 6.57 ps. Our analysis confirms that the timing closure of the critical path remains unaffected. Even under a more aggressive 2 GHz constraint, the proposed logic does not create any timing violations, ensuring that the design maintains high-frequency performance without necessitating additional pipeline stages or cycles.

Compatibility with Page Migration: cuPTW is compatible with existing page migration mechanisms. Upon initiating a migration, the GPU driver issues a flush command to invalidate the page to ensure translation coherence [6, 17, 58, 70]. Consequently, the target GPU flushes in-flight instructions in compute unit pipelines and invalidates both in-flight transactions and the states of

caches and TLBs. To support page migration, cuPTW also discards in-flight page table walks in compute units and invalidates the corresponding read-only page table entry in the L1 scalar caches. Since the invalidation of PTEs in L1 caches is performed in parallel with the TLB shootdown process, it introduces no additional latency to the migration critical path. Notably, the software-managed page walk caches remain intact, as page migration typically involves only the last-level PTEs.

Integration with On-demand Paging: cuPTW seamlessly integrates with on-demand paging placement, such as round-robin, first-touch [48, 67], and least-first [68]. When cuPTW encounters a page fault (e.g., fetching a PTE marked as invalid), it forwards the faulting address to the GMMU. The GMMU then inserts the requests into the hardware fault buffer [70], triggering an interrupt for the GPU driver. The driver subsequently resolves the fault by initiating page migration.

Extension to NVIDIA GPU: Although our design is exemplified using the AMD GPU architecture, the proposed optimizations are applicable to NVIDIA GPUs. Both architectures employ the SIMT execution model, where threads are managed at the granularity of a warp (NVIDIA) or wavefront (AMD) [1, 5, 51]. In AMD GPUs, a Compute Unit (CU) typically comprises a scalar unit, an LDS unit, four SIMD units and a branch unit. In contrast, an NVIDIA Streaming Multiprocessor (SM) consists of multiple sets of integer (INT) cores, floating-point (FP) cores, and a dedicated memory unit that manages both L1 cache and shared memory [29]. Furthermore, NVIDIA GPUs utilize a unified data cache instead of partitioned scalar and vector caches. To port our design to NVIDIA platforms, modifications are required for the execution of the translation warp (the counterpart to the translation wavefront) and the data path of cuPTW.

Specifically, the workflow is as follows: First, missed translation requests are forwarded to the SM's register file, where metadata is written to the registers of an available translation warp ("Request Setup" stage). Subsequently, when the warp scheduler issues the translation warp, the active threads are dispatched to idle INT units to calculate the physical address of the next-level PTE. During the write-back stage, results are committed to registers, and the level field is updated ("Page Offset Calculation" stage). In the next "Sent to Memory" stage, the warp is issued to the memory unit to generate a memory request. Similarly, the L1 data cache can bypass the L1 TLB and directly access cache blocks by utilizing pre-calculated physical addresses. Once the target PTE is retrieved and stored in the warp's registers, the warp is dispatched to the INT core again to determine whether the page walk has completed by comparing the level field ("Termination Check" stage). Finally, threads that have completed the walk are marked as inactive, and the obtained PTEs are populated back into the L2 TLB via the translation data path ("Translation Done" stage).

9 Related Work

Address translation has become a critical performance bottleneck in modern GPUs, and prior work has explored techniques to mitigate translation overheads from different perspectives. What sets cuPTW apart from prior work is that it re-purposes underutilized computational and memory resources to enhance page table walk throughput, effectively exploiting massive thread-level parallelism while incurring negligible hardware overhead.

A number of prior studies have sought to alleviate GPU address translation bottlenecks by improving TLB efficiency and exploiting translation reuse. Several works leverage page sharing behavior to reduce miss latency and contention [18, 24, 40, 41, 43]. B. Kotra et al. [34] and Jaleel et al. [30] enhance TLB reach by repurposing idle GPU memory resources to cache translation entries more effectively. In addition, several works have proposed some large and flexible-sized pages to increase TLB reach and consequently reduce TLB miss rate [10, 27, 37, 55]. While these efforts effectively improve TLB efficiency, they largely overlook the fundamental limitation of the GPU address translation system, i.e., the limited throughput of the page table walk process itself.

Several studies have proposed novel techniques to improve page table walk efficiency [11, 25, 36, 53, 59, 61, 62]. Shin et al. [62] leverage a single cache line to serve multiple pending translation requests in the page walk queue. Lee et al. [36] further improve page walk throughput by enabling each page walker to service multiple translation requests concurrently. Shin et al. [61] and Ausavarungnirun et al. [11] reduce page table walk overheads by scheduling walk requests and prioritizing memory accesses for translation. In addition, Park et al. [53] propose a speculative mechanism that allows memory accesses to run ahead of address translation, overlapping the two processes. In MCM-GPU systems, Barre Chord [25] improves translation efficiency through intelligent page placement across GPU chiplets.

10 Conclusion

The overhead of address translation remains a critical bottleneck for virtual memory in modern GPUs. To mitigate this, we proposed Compute Unit Page Table Walk (cuPTW), a novel architecture that re-purposes idle GPU compute resources to accelerate page table walks. cuPTW transforms the page table walk into a massively parallel computation task by leveraging underutilized SIMD lanes and a software-managed page walk cache in LDS memory to substantially boost translation throughput. Our evaluation reports that cuPTW-Full achieves a 1.97× to 2.08× over state-of-the-art GPU translation proposals by substantially improving page table walk throughput.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful feedback, which greatly enhanced the quality of this paper. This work was in part supported by the National Natural Science Foundation of China (No. 62402411), the CCF-Phytium Fund (No. CCF-Phytium202302), and the Guangdong Provincial Project (No. 2023QN10X252). Corresponding author: Jiayi Huang.

References

[1] 2024. "AMD Instinct MI300" Instruction Set Architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf.

[2] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A Badawy. 2022. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. Ieee, 1–8.

[3] Advanced Micro Devices, Inc. [n. d.]. *AMD Stream Computing Guide / ATI Stream SDK OpenCL Programming Guide*. Advanced Micro Devices, Inc. http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf Accessed: 2025-10-14.

[4] Advanced Micro Devices, Inc. 2016. *Graphics Core Next, Generation 3 Instruction Set Architecture: Reference Guide*. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/gcn3-instruction-set-architecture.pdf Accessed: 2025-10-14.

[5] Advanced Micro Devices, Inc. 2025. *Introducing AMD CDNA™4 Architecture: Breakthrough AI and HPC Acceleration with Enhanced AI Capabilities, Advanced Precisions, and High Efficiency*. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-4-architecture-whitepaper.pdf White paper.

[6] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W. Keckler, and Thomas F. Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 354–365. doi:10.1109/HPCA.2015.7056046 ISSN: 2378-203X.

[7] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 515–528. doi:10.1109/ISCA45697.2020.00050

[8] AMD. 2021. Tools and SDKs. https://developer.amd.com/amdaccelerated-parallel-processing-app-sdk/

[9] 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.

[10] 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.

[11] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. 2018. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. *ACM SIGPLAN Notices* 53, 2 (2018), 503–518.

[12] Todd M Austin and Gurindar S Sohi. 1996. High-bandwidth address translation for multiple-issue processors. In *Proceedings of the 23rd annual international symposium on computer architecture*. 158–167.

[13] Matthias Bach, Matthias Kretz, Volker Lindenstruth, and David Rohr. 2011. Optimized HPL for AMD GPU and multi-core CPU usage. *Computer Science-Research and Development* 26, 3 (2011), 153–164.

[14] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 163–174.

[15] Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2022. Navisim: A highly accurate gpu simulator for amd rdna gpus. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 333–345.

[16] 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 (June 2010), 48–59. doi:10.1145/1816038.1815970

[17] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A Mojumder, José L Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-software support for efficient page migration in multi-gpu systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 596–609.

[18] 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*. ACM, Virtual Event GA USA, 455–466. doi:10.1145/3410463.3414639

[19] Abhishek Bhattacharjee. 2013. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 383–394.

[20] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 141–151.

[21] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonia. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 128–139.

[22] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 185–195.

[23] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*. 63–74.

[24] Yajuan Du, Mingyang Liu, Yuqi Yang, Mingzhe Zhang, and Xulong Tang. 2022. Enhancing GPU performance via neighboring directory table based inter-TLB sharing. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 146–153.

[25] Yuan Feng, Seonjin Na, Hyesoon Kim, and Hyeran Jeon. 2024. Barre Chord: Efficient Virtual Memory Translation for Multi-Chip-Module GPUs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 834–847. doi:10.1109/ISCA59077.2024.00065

[26] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*. 224–235.

[27] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. 2023. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, Vancouver BC Canada, 433–448. doi:10.1145/3582016.3582021

[28] Mark Harris. 2017. *Unified Memory for CUDA Beginners*. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ Accessed: Oct. 14, 2025.

[29] Rodrigo Huerta, Mojtaba Abaie Shoushtary, José-Lorenzo Cruz, and Antonio Gonzalez. 2025. Dissecting and modeling the architecture of modern GPU cores. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 369–384.

[30] 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.
[31] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Virtual Event Greece, 738–753. doi:10.1145/3466752.3480063
[32] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. 2023. Victima: Drastically increasing address translation reach by leveraging underutilized cache resources. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1178–1195.
[33] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G Rogers. 2020. Locality-centric data and threadblock management for massive GPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1022–1036.
[34] Jagadish B Kotra, Michael LeBeane, Mahmut T Kandemir, and Gabriel H Loh. 2021. Increasing gpu translation reach by leveraging under-utilized on-chip resources. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1169–1181.
[35] David Kroft. 1998. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*. 195–201.
[36] Jiwon Lee, Gun Ko, Myung Kuk Yoon, Ipoom Jeong, Yunho Oh, and Won Woo Ro. 2025. Marching Page Walks: Batching and Concurrent Page Table Walks for Enhancing GPU Throughput. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1662–1677.
[37] 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.
[38] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*. IEEE, 260–271.
[39] 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.
[40] 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.
[41] Bingyao Li, Yueqi Wang, Tianyu Wang, Lieven Eeckhout, Jun Yang, Aamer Jaleel, and Xulong Tang. [n. d.]. STAR: Sub-Entry Sharing-Aware TLB for Multi-Instance GPU. ([n. d.]).
[42] 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.
[43] Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. 2021. Improving address translation in multi-gpus via sharing and spilling aware tlb design. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1154–1168.
[44] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and dissecting the nvidia hopper gpu architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 656–667.
[45] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Vol. 213. 1.
[46] Michael J Mantor and Brian Emberling. 2017. SIMD processing unit with local data share and access to a global data share of a GPU. US Patent 9,619,428.
[47] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th annual international symposium on Computer architecture*. 235–246.
[48] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Cambridge Massachusetts, 123–135. doi:10.1145/3123939.3124534
[49] Prabhat Mishra, Nikil Dutt, and Alex Nicolau. 2001. A study of out-of-order completion for the MIPS R10K superscalar processor. (2001).
[50] NVIDIA. 2016. *GPU-Accelerated Applications*. http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf Accessed: Oct. 15, 2025.

[51] NVIDIA Corporation. 2025. *CUDA C++ Programming Guide*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ Accessed: 2025-10-14.
[52] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every walk'sa 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.
[53] Junhyeok Park, Osang Kwon, Yongho Lee, Seongwook Kim, Gwangeun Byeon, Jihun Yoon, Prashant J. Nair, and Seokin Hong. 2024. A Case for Speculative Address Translation with Rapid Validation for GPUs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 278–292. doi:10.1109/MICRO61859.2024.00029 ISSN: 2379-3155.
[54] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic resource management for efficient utilization of multitasking GPUs. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 527–540.
[55] 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.
[56] 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.
[57] Louis-Noel Pouchet and Tomofumi Yuki. 2010. Polybench. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[58] 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.
[59] 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.
[60] Bharadwaj 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.
[61] 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.
[62] 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.
[63] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. 2019. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*. 197–209.
[64] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.
[65] Vinay Vashishtha, Manoj Vangala, and Lawrence T Clark. 2017. ASAP7 predictive design kit development and cell design technology co-optimization. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 992–998.
[66] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 161–171.
[67] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 339–351.
[68] 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, Volume 2*. ACM, Vancouver BC Canada, 544–559. doi:10.1145/3575693.3575745
[69] Xia Zhao, Guangda Zhang, Lu Wang, Shiqing Zhang, and Huadong Dai. 2025. NearFetch: Saving Inter-Module Bandwidth in Many-Chip-Module GPUs. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1693–1706.
[70] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.

Received January 2026; revised March 2026; accepted April 2026