NB-Walker: Non-Blocking Page Table Walkers to Enhance Address Translation in NUMA GPUs

Zihang Chen*, Lieven Eeckhout‡, Hongyuan Liu§, Tianao Ge¶⋆, Xinkai Wang‖, Jiayi Huang*††
*The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China
†The Hong Kong University of Science and Technology, Hong Kong SAR, China
‡Ghent University, Ghent, Belgium
§Stevens Institute of Technology, Hoboken, USA
¶Shenzhen University of Advanced Technology, Shenzhen, China
‖Shanghai Jiao Tong University, Shanghai, China
Emails: zchen097@connect.hkust-gz.edu.cn, lieven.eeckhout@UGent.be, hliu96@stevens.edu
getianao@suat-sz.edu.cn, unbreakablewxk@sjtu.edu.cn, hjy@hkust-gz.edu.cn

***Abstract*—Modern Graphics Processing Unit (GPU) architectures have transitioned toward multi-partitioned or chiplet-based designs to scale compute throughput and memory capacity. However, this evolution introduces significant internal Non-Uniform Memory Access (NUMA) effects, where cross-partition memory latencies diverge significantly. As address translation requires sequential, multi-level probes, these NUMA-induced latencies are amplified, placing unprecedented pressure on the memory management unit. This paper identifies a critical bottleneck: traditional page table walkers employ a rigid, blocking design, where walker hardware resources are held by in-flight requests until the entire multi-level walk completes. Under NUMA conditions, long-latency remote memory probes monopolize these resources, causing translation requests to queue, which eventually stalls the entire GPU pipeline.**

To mitigate these translation-induced stalls, we propose NB-WALKER, a high-throughput, non-blocking address translation architecture. NB-WALKER decouples translation issuance from limited page walk buffers by repurposing underutilized L1 cache MSHR entries to buffer walk metadata and track outstanding page table entry fetch requests. This transformation allows the page table walkers to sustain multiple outstanding translation requests concurrently, significantly enhancing memory-level parallelism during the walk process. Furthermore, our design enables opportunistic coalescing of concurrent page table probes, reducing redundant memory traffic to the last-level cache. Experimental results demonstrate that NB-WALKER significantly alleviates translation pressure, achieving an average 5.56× increase in page table walk throughput. This enhanced concurrency translates into an average speedup of 2.78× (up to 5.73×) across translation-intensive workloads with negligible hardware overhead.

***Index Terms*—Graphics Processing Unit, Virtual Memory, Address Translation, Page Table Walker**

I. INTRODUCTION

GPUs have become indispensable for high-performance computing and machine learning, driven by their massive thread-level parallelism and superior memory bandwidth [1]–[7]. To meet the surging demands for compute throughput and memory capacity, modern GPU architectures, such as the Nvidia A100 and H100, have evolved into multi-partitioned designs connected via a network-on-chip (NoC). While this discrete organization enables scaling of the number of cores (e.g., streaming multiprocessors) and Last-Level Cache (LLC) slices, it inherently introduces a NUMA effect. This spatial distribution creates significant latency disparities between local and cross-partition memory accesses, posing a new challenge to performance scalability [8]–[10].

The performance impact of internal NUMA effects is further compounded by the evolving memory management paradigms. Traditionally, GPUs suffered from complex programming models and significant data movement overheads [11]. To mitigate these challenges, vendors have introduced a unified address space, commonly realized as Unified Virtual Memory (UVM). While UVM provides a seamless programming abstraction by enabling transparent pointer dereferencing, it creates a logical illusion of memory homogeneity that contradicts the underlying NUMA reality. Consequently, the efficiency of UVM becomes heavily contingent upon the underlying address translation mechanism. Despite its programmability benefits, address translation frequently emerges as a primary performance bottleneck, especially for applications characterized by irregular memory access patterns or massive memory footprints [12]–[18].

To alleviate the severe bottleneck of address translation, modern GPU architectures incorporate specialized hardware support, such as hierarchical Translation Lookaside Buffers (TLBs) for caching frequent mappings alongside highly threaded walkers to facilitate concurrent page table walks [15], [17], [22], as shown in Figure 1. However, for applications exhibiting frequent last-level TLB (i.e., L3 TLB) misses, the surge of translation misses often exceeds the processing capacity of the Graphics Memory Management Unit (GMMU). Due to the insufficient parallelism of page table walks in the GMMU, these pending requests are forced to queue for thousands of cycles [16], [23]. As a result, unresolved misses rapidly saturate the limited Miss Status Holding Register

†Corresponding author.
⋆This work was completed while the author was affiliated with the Hong Kong University of Science and Technology (Guangzhou), China.

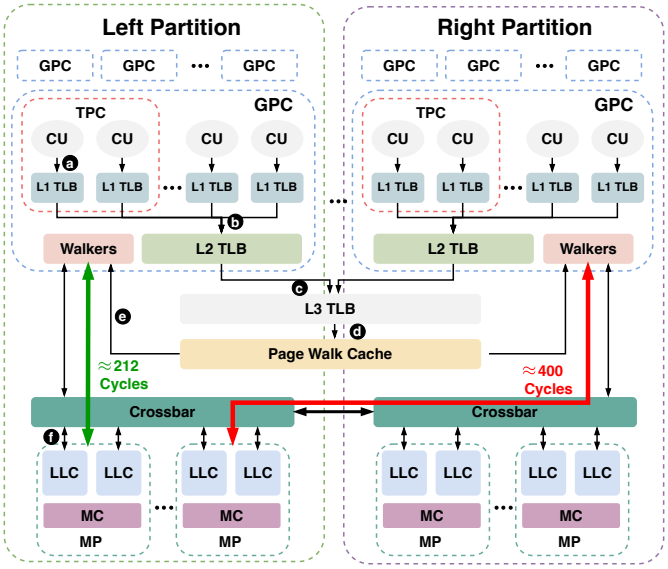


Fig. 1: TLB structure and address translation process in modern GPUs [8], [19]–[21]. Walkers follow a *blocking* design that keeps them occupied until each walk completes, while PTW memory accesses suffer from *non-uniform* latency due to internal NUMA effects.

(MSHR) resources, triggering back pressure that stalls the entire address translation subsystem, thereby leading to a severe degradation in overall GPU throughput [15]–[17], [23], [24].

Prior research has extensively explored optimizing the Page Table Walk (PTW) mechanism to enhance effective translation throughput. For instance, intelligent scheduling [17], [24] prioritizes critical PTW requests to minimize GPU stalls, while neighborhood-aware translation [15] and Marching Page Walks (MPW) [16] leverage spatial locality to batch multiple translation probes. In addition, two concurrent efforts, SoftWalker [23] and cuPTW [25], exploit underutilized compute resources to perform page table walks, improving PTW throughput through massive thread-level parallelism. Despite their advancements, these mechanisms remain fundamentally *blocking*: resources allocated to a PTW request, such as hardware page walk buffer entries in conventional designs or software translation threads in SoftWalker and cuPTW, remain occupied until the entire multi-level walk completes. This leads to severe resource under-utilization during long-latency memory probes.

The blocking nature of the page table walk hardware is a critical bottleneck in modern large-scale GPUs with a NUMA memory subsystem. Prior PTW designs predominantly assume a uniform memory model, however in a partitioned GPU, the latency of a PTW is highly sensitive to the physical location of the Page Table Entries (PTEs). Specifically, when a translation request triggers a remote-partition memory access via the NoC, the walker incurs substantial cross-partition latencies as shown in Figure 1. Under a blocking paradigm, these delayed requests monopolize critical translation resources (such as the page walk buffer entries shown in Figure 2) for a long duration. This increases the occupancy time of each walker, which in turn triggers a cascade of queueing delays for subsequent requests [23]. Consequently, the NUMA effect compounds the blocking bottleneck, negating the parallelism gains of state-of-the-art walkers and causing the translation to become the primary performance bottleneck in memory-intensive workloads.

To address these bottlenecks, we conduct a comprehensive characterization of the system, yielding the following key insights: (1) *Latency-throughput trade-off*: Our analysis reveals that prioritizing address translation throughput at the cost of a marginal increase in page table walk latency yields significant overall performance gains for memory-intensive workloads. (2) *L1 MSHR Underutilization:* For translation-intensive workloads, we observe that L1 MSHRs consistently exhibit low utilization, suggesting an opportunity to repurpose these resources for more efficient walk metadata management. (3) *Non-uniform memory access latency during PTW:* Through detailed breakdown of page table walk overheads, we identify a pronounced NUMA effect in applications with high walk frequency. Specifically, memory access latencies follow a bimodal distribution, driven by the physical proximity between walkers and target memory partitions.

Motivated by these observations, we propose NB-WALKER, a *non-blocking* address translation architecture that exploits idle L1 MSHR entries to track and buffer metadata for PTW requests. Unlike traditional walkers that often stall the walk pipeline while waiting for memory responses, NB-WALKER leverages the MSHR's capability to manage multiple outstanding requests in parallel. By transforming the PTW process into a non-blocking operation, our design significantly enhances PTW throughput and Memory-Level Parallelism (MLP) with negligible hardware overhead. Furthermore, this approach increases coalescing opportunities for LLC accesses, effectively reducing redundant memory traffic and mitigating the translation bottleneck. We make the following contributions:

- We characterize and quantify the impact of NUMA effects, arising from increasingly large GPU die sizes, on page table walks.
- We observe a critical bottleneck where low PTW throughput restricts the effective utilization of L1 caches. Stalled translation requests prevent the core from issuing sufficient memory demands, leading to under-utilization of L1 MSHRs.
- We propose NB-WALKER, a novel architecture that decouples page walk logic from the physical storage within the walker. By repurposing idle L1 MSHRs to buffer PTW metadata and track PTE fetch memory requests, NB-WALKER eliminates the structural blocking inherent to traditional walkers. Experimental results show that NB-WALKER improves PTW throughput by 5.56× on average, resulting in a 2.78× performance speedup for translation-intensive workloads with negligible hardware overhead.

II. Background

A. GPU Non-Uniform Memory Access

Modern GPUs leverage a Single-Instruction, Multiple-Thread (SIMT) execution model to achieve massive computational throughput [26], [27]. At the hardware level, a GPU is organized into multiple Compute Units (CUs) — in AMD terminology — or Streaming Multiprocessors (SMs) — in Nvidia terminology. Each unit features a private L1 cache and dedicated high-bandwidth shared memory. In addition, a global L2 cache is partitioned into multiple slices and distributed near the memory controllers. To sustain the massive throughput required by modern workloads, contemporary GPU architectures have aggressively scaled their on-chip resources. For instance, the Nvidia H100 GPU integrates 132 SMs alongside a highly partitioned memory subsystem, featuring 80 LLC slices and 10 memory controllers [8]. To efficiently orchestrate massive hardware resources, the H100 employs a hierarchical architecture. On the compute side, SMs are organized into Texture Processing Cluster (TPCs) and further aggregated into Graphics Processing Clusters (GPCs). Symmetrically, the memory subsystem is partitioned into clusters of LLC slices and memory controllers, i.e., memory partitions (MPs). These resource clusters are interconnected via a Converge-Diverge Crossbar network, facilitating high-bandwidth communication across the entire chip. Furthermore, recent GPUs, including the Nvidia Ampere (A100), Hopper (H100), and Blackwell (B100) and AMD CDNA 3 architecture (e.g., MI300A and MI300X) [9], [10], [28]–[30] adopt partitioned or multi-die designs to scale capacity. These architectures introduce a pronounced NUMA effect, where the latency and bandwidth of data movement varies significantly depending on the physical distance between compute units and memory partitions [8]–[10]. As illustrated in Figure 1, we derive our baseline timing parameters from the empirical study by Jin et al. [8], who reverse-engineered the memory hierarchy of the Nvidia A100 and H100 GPUs through fine-grained microbenchmarking. They report that requests from GPCs 0–3 to their local MP0 memory partition incur a hit latency of about 212 cycles, whereas requests from the remote GPCs 4–7 to the same MP0 require roughly 400 cycles. These two latency values are used to model local and remote NUMA accesses in our baseline system, as described in Section V-A1.

B. Address Translation in GPUs

Figure 1 illustrates the GPU address translation hierarchy. Each memory request performs a parallel lookup of the Virtually-Indexed Physically-Tagged (VIPT) L1 data cache and private L1 TLB (ⓐ). Upon an L1 TLB miss, the request checks the MSHR, which coalesces redundant misses to the same page and issues an outstanding translation request to the GPC-shared L2 TLB (ⓑ). Similarly, an unresolved request further propagates to the GPU-wide L3 TLB (ⓒ). Upon an L3 miss, the page walk subsystem intercepts the request at the Page Walk Cache (PWC) (ⓓ) to retrieve intermediate page directory entries, potentially skipping levels in the 4-level radix page table. The PTW request is subsequently dispatched to the available GPC-shared page walkers (ⓔ) to traverse the multi-level radix page table resident in the memory partitions.

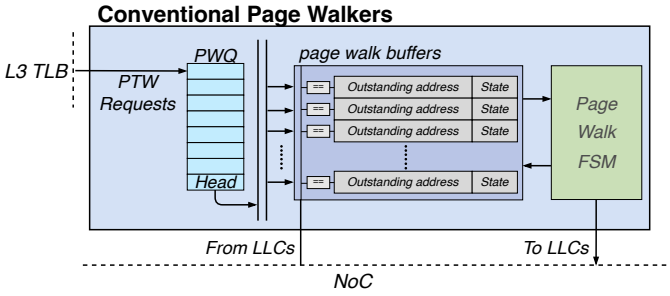


Fig. 2: Overview of page walkers in the baseline GPU.

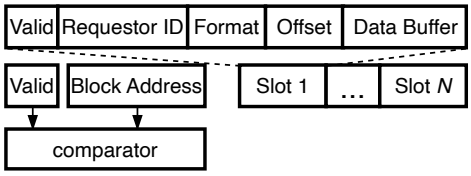


Fig. 3: Overview of an explicitly addressed MSHR entry.

As shown in Figure 2, each walker comprises a Page Walk Queue (PWQ), a Content-Addressable Memory (CAM) based Page Walk Buffer (PWB), and a Page Walk Finite State Machine (FSM). Dispatched requests are buffered in the PWQ and allocated to the PWB as entries become available. The FSM then calculates the physical address of the next-level PTE and issues memory requests to the LLC (ⓕ in Figure 1). Upon the PTE's return, the PWB performs a CAM-based match against all outstanding requests to identify the corresponding entry. The FSM then evaluates the PTE to either proceed to the next level or terminate the walk. Any translation failure (e.g., invalid entries or remote page access) is propagated as a page fault to the host-side IOMMU via a dedicated fault buffer [31]. Finally, the terminal PTE populates the L3 TLB and resolves the pending MSHR.

C. Explicitly Addressed MSHR in GPUs

Modern GPU caches typically employ a non-blocking design to sustain high memory throughput. This architecture primarily comprises a tag/data array for caching frequently accessed blocks and MSHRs [32], [33] for tracking outstanding misses. While the data array maintains frequently accessed data to leverage temporal locality, the MSHRs track outstanding cache misses to enable non-blocking cache operation. To exploit spatial locality, data transfers between cache hierarchies are performed at the granularity of a cache line (e.g., 64 bytes). Upon an L1 cache miss, the cache controller allocates an entry in the MSHR to track the request before subsequently issuing a transaction to the L2 cache. While the transaction is pending, subsequent misses targeting the same cache line are identified through a CAM lookup, resulting in an MSHR hit. These requests are then coalesced into the existing

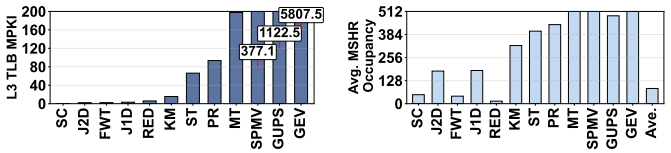


Fig. 4: L3 TLB MPKI (left) and Average L3 TLB MSHR occupancy (right) on the baseline GPU .

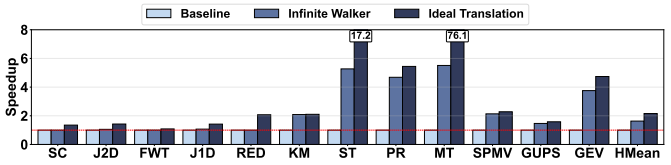


Fig. 5: Performance speedup for baseline GPU, infinite page walkers, and ideal translation configuration.

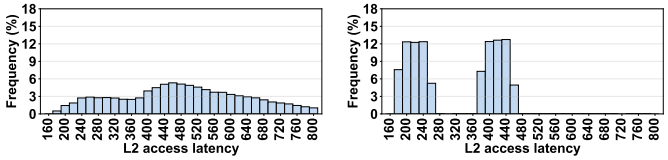
MSHR entry, thereby eliminating redundant memory traffic and maximizing MSHR utilization efficiency. Farkas et al. [33] proposed the explicitly addressed MSHR, which is widely adopted in modern GPUs. As shown in Figure 3, an explicitly addressed MSHR entry typically comprises two components: comparison logic and multiple MSHR slots. Each slot captures the essential attributes of a merged request, including the requester ID, request format, and intra-cacheline offset.

III. Motivation

A. Overall Performance Characterization

In this section, we quantitatively characterize the benchmarks on the baseline GPU architecture using the industrial-validated MGPUSim simulator [34]. Details of the configuration and applications are provided in Section V-A. Our analysis focuses on two critical dimensions that significantly impact system performance: (1) the bottlenecks of address translation in modern GPUs, and (2) the impact of NUMA effects during the page walk process.

The Address Translation Bottleneck. We first characterize the translation pressure across the evaluated benchmarks and categorize them into two groups based on their L3 TLB Misses-Per-Kilo-Instructions (MPKI): low-MPKI applications and high-MPKI applications (detailed in Section V-A2). As illustrated in Figure 4 (left), the workloads exhibit a broad spectrum of translation intensity: the L3 TLB MPKI ranges from 0.40 in SC to over 5,000 in GEV. To quantify the sensitivity of GPU performance to PTW throughput, we evaluate three configurations: (1) a baseline GPU, (2) a GPU with unconstrained PTW throughput (idealized as infinite walkers), and (3) a GPU with ideal address translation (zero-latency). As shown in Figure 5, page table walk throughput exerts a negligible influence on overall performance for low-MPKI applications, such as SC and FWT). In contrast, it becomes a critical performance bottleneck for workloads exhibiting a high L3 TLB MPKI, such as MT and ST. In such scenarios, frequent L3 TLB misses generate a surge of PTW requests that saturate



(a) FWT (low-MPKI) (b) MT (high-MPKI)

Fig. 6: L2 latency histogram distribution of page table walks for two representative benchmarks.

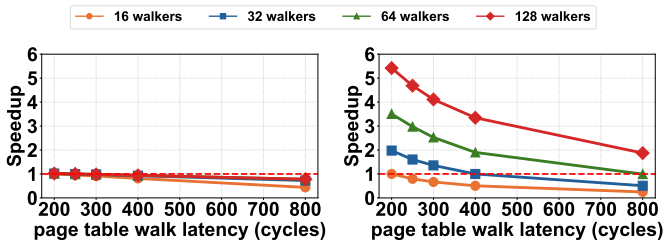
the page walk queue and incur thousands of cycles queueing latency [16]. This congestion incurs two primary performance degradation mechanisms. First, pending TLB misses resident in the L3 TLB MSHRs cannot be resolved promptly, eventually leading to MSHR exhaustion that back-pressures and blocks subsequent TLB lookups. Figure 4 (right) illustrates the average L3 TLB MSHR occupancy on the baseline GPU. For workloads characterized by frequent L3 TLB misses (e.g., KM, MT, and ST), the L3 TLB MSHRs remain persistently saturated. Second, the substantial translation latency prevents dependent cache requests from obtaining physical addresses, thereby stalling the tag-matching process and halting further memory operations.

How NUMA Affects PTW. To investigate the impact of NUMA effects on GPU page table walks, we characterize the latency distribution of page table walks accessing LLC slices on our baseline GPU, using FWT (low-MPKI) and MT (high-MPKI) as representative benchmarks. For FWT, the PTW latency is dominated by off-chip DRAM access (>400 cycles), as infrequent PTE requests fail to exploit temporal locality in the LLC, see Figure 6(a). While these latencies are high, the low intensity of page walks ensures that they do not reside on the critical path. In contrast, frequent L3 TLB misses in MT trigger intensive page table walks that exhibit high temporal locality. These frequently accessed PTEs are persistently cached within the distributed LLCs. Therefore, the NUMA effect exerts a substantial impact on the latency characteristics of MT, as the on-chip interconnect overhead becomes the primary bottleneck. As illustrated in Figure 6(b), the LLC access latencies primarily exhibit a distinct bimodal distribution: one ranging from 180 to 260 cycles and the other from 380 to 460 cycles.

Implication #1: NUMA effects significantly influence the page table walk behavior of high-MPKI workloads.

B. Conventional GMMU Characterization

To quantify the impact of page table walks on overall GPU performance, we conduct a comprehensive characterization study. Figure 7 illustrates the sensitivity of diverse GPU workloads to both PTW throughput and latency. As depicted in Figure 7(a), applications with low MPKI exert lower pressure on the page table walker due to infrequent L3 TLB misses, thereby demonstrating negligible sensitivity to



(a) low-MPKI applications (b) high-MPKI applications

Fig. 7: Performance speedup for (a) low-MPKI and (b) high-MPKI applications across varying page table walker counts and walk latencies, normalized to a baseline with 16 walkers and a 200-cycle walk latency.

both PTW throughput and latency. In contrast, as illustrated in Figure 7(b), the performance of high-MPKI applications heavily relies on the efficiency of the page table walker. In these cases, excessive walk latency leads to stalls induced by serialized page walks, which emerge as a primary performance bottleneck. However, our characterization further reveals that increasing the page walk throughput allows the overhead of prolonged latencies to be effectively hidden by highly concurrent PTW requests, e.g., 128 walkers with a 400-cycle page table walk latency outperform 16 walkers with a 200-cycle walk latency by nearly 3×.

Implication #2: Increases in page table walk latency are tolerable if they are compensated for by a substantial gain in concurrency.

To accelerate address translation, conventional GPUs employ a threaded page table walker within the GMMU. This walker utilizes a finite state machine to perform the walking process while relying on page walk buffers to track in-flight memory requests [35], as illustrated in Figure 2. To facilitate rapid state updates, the buffer is implemented as CAM, enabling the walker to match returning page table data with the corresponding pending request [16], [35]. However, despite this efficient lookup mechanism, the threaded page table walker is still characterized as a '*blocking*' walker. Each request occupies a buffer entry from the initial miss until the completion of the entire multi-level walk, thereby imposing a structural bottleneck that limits the number of concurrent translations. A straightforward approach to scaling PTW parallelism is to increase the capacity of the page walk buffers. Expanding these structures provides the necessary entries to support more parallel walkers, allowing the walkers to drain the page walk queue more efficiently. However, to sustain higher PTW parallelism, the port count of the CAM must be scaled to bolster throughput. Such scaling results in prohibitive hardware area overhead, as the area complexity of multi-ported CAM cells scales quadratically with the number of ports [36]. Specifically, Lee et al. [16] report that for the Nvidia V100 GPU, at least a 9.1× area expansion within the

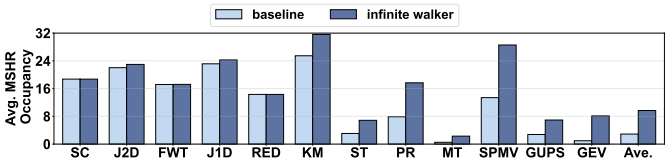


Fig. 8: Average L1 MSHR occupancy across different benchmarks under baseline GPU and idealized configuration with infinite walkers.

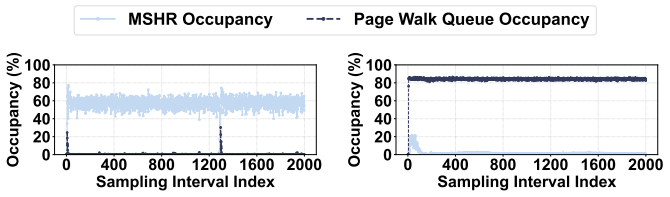
page table walker is required to achieve peak performance.

Implication #3: A conventional GMMU employs threaded blocking page table walkers, which limits the parallelism of PTW requests.

C. Under-Utilized On-Chip Memory Resources

Modern GPU architectures feature a multi-level memory hierarchy, comprising private L1 caches per compute unit, a globally shared L2 cache, and high-bandwidth, high-latency VRAM (e.g., GDDR or HBM). Distinct from the shared L2 and VRAM, these dedicated L1 caches offer significantly lower latency and superior bandwidth [28], [29]. To investigate the potential for better leveraging these dedicated L1 caches, we evaluate the average occupancy of L1 MSHR entries across various benchmarks under both the baseline GPU and an idealized configuration with infinite walkers, see Figure 8. For low-MPKI applications (e.g., SC and FWT), due to the low translation pressure, the MSHR occupancy of the baseline GPU remains consistent with the idealized configuration, which indicates that address translation is not on the critical path for these workloads. In contrast, for high-MPKI applications (e.g., PR and SPMV), the MSHR occupancy under the idealized configuration is significantly higher compared to the baseline. This disparity suggests that the low MSHR utilization in the baseline stems from severe translation pressure. Specifically, when the address translation becomes a bottleneck due to frequent L3 TLB misses, L1 caches must stall for thousands of cycles awaiting translation responses. During these stalls, the cache is unable to proceed with tag lookups for pending memory requests, which consequently throttles MSHR allocation and underutilizes the available memory resources.

Furthermore, to evaluate structural pressure, we select one representative application from each of the two categories to analyze the temporal occupancy trends of the L1 cache MSHRs and PWQ. We sample the average occupancy at a granularity of 1,000 cycles. As shown in Figure 9(a), FWT maintains moderate MSHR occupancy and minimal PWQ pressure. This is attributed to its high spatial locality and inter-thread data sharing, which reduces the frequency of unique page table walks. Figure 9(b) illustrates the structural occupancy trends for MT. As the PTW subsystem becomes a bottleneck under intense translation pressure, the L1 caches are frequently stalled while awaiting physical address resolution,



(a) FWT (b) MT

Fig. 9: Dynamic resource occupancy of L1 MSHRs vs. PWQs. Sampling over 1,000-cycle intervals.

preventing the MSHRs from being effectively utilized. Consequently, consistent with the aggregate statistics in Figure 8, L1 MSHR occupancy remains persistently low while PWQ reaches saturation.

Implication #4: For applications with high MPKI, the address translation bottleneck often underutilizes the L1 data cache, as the compute units are frequently stalled awaiting TLB resolutions.

IV. Non-Blocking Page Table Walker

Contemporary GMMUs employ multiple page table walkers to handle concurrent PTW requests. Although these walkers can traverse page tables in parallel, each individual page table walker is fundamentally **blocking**: *it must wait for the memory subsystem to return the current PTE before proceeding*. This rigid coupling between PTW request processing and memory access restricts each walker to servicing a single request at a time, failing to exploit potential MLP within the GMMU. To address the limitations of blocking page table walkers, we propose a novel non-blocking walker architecture, NB-Walker, that decouples request processing from memory accesses, thereby improving MLP and translation throughput.

A. Overview of Non-blocking Page Table Walks

1) Design goals: The primary goal of NB-Walker is to enhance translation throughput by breaking rigid dependencies in traditional GMMUs, guided by the following two primary objectives:

Decoupling Address Generation from Memory Service. Conventional GMMUs tightly couple request issuance with the PWB, which stores metadata of all outstanding memory requests. This coupling forces the walker to stall upon buffer saturation or long memory latencies, limiting MLP and overall walk throughput. Our design decouples request generation from buffer management, enabling the page walk FSM to process new PTW requests independently of pending access latencies.

Maximizing PTW Concurrency via Idle L1 Cache MSHRs. Decoupling requires a hardware structure capable of both metadata buffering and efficient CAM-based searching. As analyzed in Section III-C, L1 cache MSHRs are often underutilized under intensive translation pressure, and their native CAM structure is well-suited for our requirements. Since the L1 cache and page walker are co-located within the same GPC, they offer high-bandwidth, low-latency communication. By offloading in-flight PTW metadata to these idle MSHR entries, our design achieves significantly higher concurrency than traditional coupled structures.

2) Hardware changes: To support our proposed NB-Walker, we introduce minimal yet strategic modifications to the existing GPC architecture as summarized below:

PWQ Partitioning. We split the baseline PWQ into a 28-entry page walk request queue (PWReqQ) and a 16-entry page walk response queue (PWRspQ). The PWReqQ buffers 48-bit incoming requests, while the PWRspQ captures 42-bit metadata (40-bit VPN, 2-bit level) and 8-byte PTEs. This asymmetric sizing minimizes SRAM area overhead while maintaining throughput.

FSM Input Multiplexer. To integrate decoupled components (i.e., PWReqQ, PWRspQ, PWB and FSM) into the page walkers, we introduce a 3-to-1 multiplexer (Mux) at the input of the page walk FSM. This Mux arbitrates among three sources: (1) new requests from the PWReqQ head, (2) metadata and PTEs from the PWRspQ head, and (3) buffered states from the PWB.

Intra-GPC Interconnect. To support the data path between the page walker and the L1 cache, we implement a bidirectional interface at the GPC-level Mux using a 2-to-1 mux/demux pair. This steers traffic between the page walker and the L1 cache, ensuring seamless integration into the cache hierarchy with minimal interconnect overhead.

L1 Cache Extension. To facilitate the direct injection of PTW metadata from the walkers into the L1 cache MSHR, we implement a lightweight datapath extension within the L1 controller. Specifically, a 2-to-1 Mux is integrated into the MSHR allocation logic, providing a dedicated bypass for PTW-issued memory requests. This path allows PTW metadata to populate MSHR entries directly, circumventing tag-matching and cache-allocation pipelines. By positioning this logic outside the primary hit/miss critical path, the extension ensures negligible impact on the baseline L1 latency. Additionally, each MSHR entry is augmented with a 1-bit PTW flag to indicate whether it is currently managing a page table walk. Since page table pages are strictly aligned with physical page boundaries (e.g., 4 KB), a single cache line access never spans both a page table page and a data page. This alignment ensures that an MSHR entry never concurrently services both kernel data and PTW requests.

B. Workflow of Non-Blocking PTWs

In this section, we detail the workflow of our proposed NB-Walker, see also Figure 10.

Upon receiving a request from the L3 TLB, the page walkers within the GPC append the request to the PWReqQ (❶), which employs a First-Come-First-Serve (FCFS) policy to preserve temporal ordering. The issue logic at the head of the PWReqQ initiates a resource check by probing the PWB to reserve an

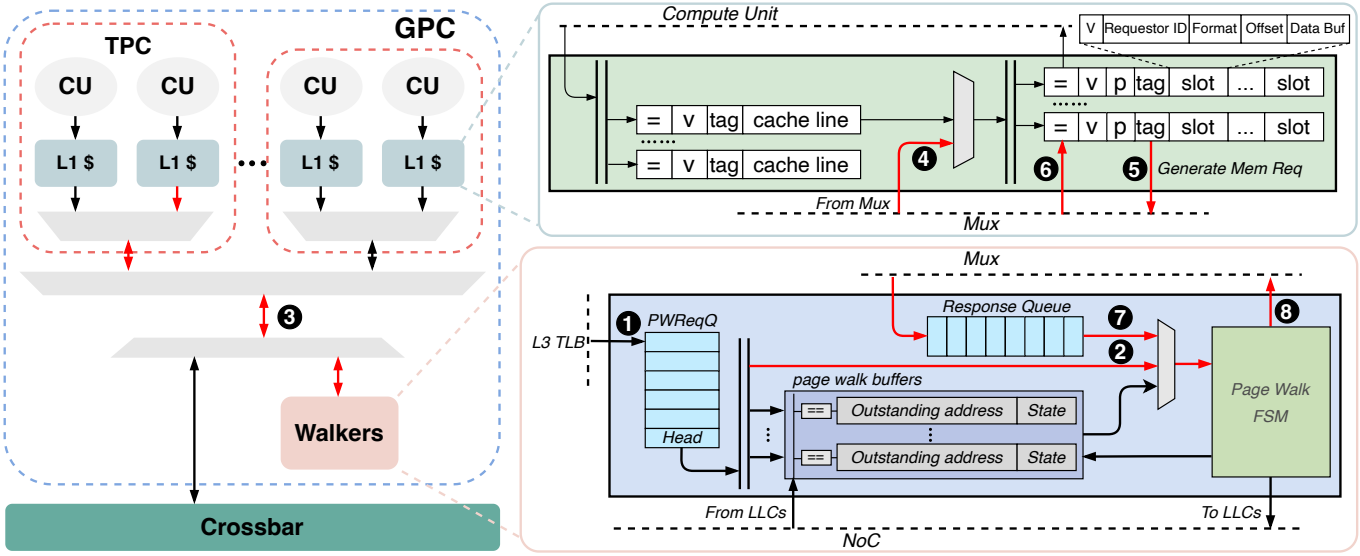


Fig. 10: Schematic overview of the NB-WALKER non-blocking page table walk subsystem.

entry. If an entry is available, the request is dispatched to the FSM via the 3-to-1 MUX for processing. The FSM then calculates the physical address of the next-level PTE, issues a memory request to the LLC, and records the outstanding address into the reserved PWB entry for subsequent address matching. Conversely, if the PWB if full, the PWReqQ head issue logic bypasses the PWB and dispatches the request directly to the FSM via the Mux (❷). While the FSM still computes the next-level PTE physical address based on the VPN, these bypassed requests follow a distinct data path. Specifically, the FSM embeds the PTW metadata (i.e., the walk level and the VPN) into the memory request payload. Subsequently, these augmented memory requests are issued to the L1 data caches within the GPC via a bidirectional Mux (❸).

Upon receiving a memory request, the L1 data cache identifies it as a PTW-initiated request via its requester ID. By leveraging a 2-to-1 Mux positioned within the L1 cache, this request bypasses the standard tag lookup and directly accesses the MSHR (❹). Similar to conventional functional unit requests, a PTW-initiated request triggers a CAM-based search across all valid MSHR entries to identify any existing entry targeting the same cache line. If a match is found, the controller reserves and initializes a free slot within that entry. Otherwise, it allocates a new MSHR entry, initializes the necessary fields, and issues a 64-byte fetch request to the LLC (❺). The initialization of an MSHR entry allocated for a PTW request follows a specialized step: the hardware first asserts a dedicated PTW flag to indicate that the entry is servicing address translation. Subsequently, a slot is allocated to store the PTW metadata. Since PTW requests are strictly 8-byte PTE read operations, the "Format" field remains constant, and the "Requester ID" is omitted as the L1 cache exclusively services local walkers. The controller only records the intra-cacheline offset and populates the data buffer with relevant metadata.

The memory requests traverse the memory hierarchy to the LLC or DRAM. Upon receiving a 64-byte data block (comprising eight 8-byte PTEs) from the LLC, the L1 cache performs a tag match in the MSHR to identify the corresponding entry (❻). The controller first checks the designated PTW flag; if asserted, it invokes a specialized processing flow for the PTW-allocated MSHR entry. Specifically, the controller processes each slot within the MSHR entry sequentially. First, it disregards the "Requestor ID" and "Format" fields, utilizing the stored offset bits to extract the target 8-byte PTE from the retrieved 64-byte cache line. Then, the controller loads the PTW metadata temporarily stored in the data buffer and concatenates it with the 8-byte PTE. This combined data block is then embedded into the memory transaction payload. Finally, the generated transaction is dispatched to the PWRspQ within the GPC.

Memory response packets returning from the L1 data cache are enqueued into the PWRspQ following a FCFS policy. Similar to the PWReqQ, the head of the PWRspQ also dispatches packets to the FSM via the 3-to-1 Mux. Then, the FSM evaluates the VPN, the retrieved PTE and the current walk level to determine if the translation process has finished (❼). If the FSM reaches a leaf-level PTE, it terminates the walk and transmits a translation response back to the L3 TLB (❽). Otherwise, the FSM calculates the address for the next page table level and generates a new memory request. Concurrently, the FSM checks for an available entry in the PWB. If a free entry is identified, the request metadata is recorded in the PWB and the request is issued directly to the LLC; otherwise, it is dispatched to the L1 cache again.

L1 Cache Routing Policy. To determine the target L1 cache for walker-initiated requests within a GPC, we evaluated

two dispatch policies: (1) XOR-based hashing derived from physical addresses [20], and (2) a Round-robin (RR) strategy. Our analysis reveals that XOR-based hashing leads to load imbalance in several workloads while offering negligible improvements in MSHR hit rates. Therefore, we adopt the RR strategy because of its superior load balancing and implementation simplicity.

Arbitration Logic for Interconnect Multiplexers. To manage data movement across the integrated paths, we implement tailored arbitration policies for each Mux. The 3-to-1 Mux at the FSM interface employs a fixed-priority scheme: requests from the PWB are prioritized as they represent the oldest in-flight operations, followed by the PWRspQ and the PWReqQ. Conversely, the bidirectional Mux within the GPC and the 2-to-1 Mux in the L1 cache utilize RR arbitration. This RR policy prevents interconnect monopolization, mitigating starvation and ensuring forward progress for all concurrent memory requests.

C. Adaptive MSHR Reservation

In NB-WALKER, MSHR entries serve a dual purpose: they function as traditional MSHRs for L1 caches and as temporary buffers for in-flight PTW metadata and coalesced PTE fetch requests. While this optimization significantly boosts translation throughput, it introduces resource contention on the MSHRs. We observed that for some memory-intensive applications with high L1 cache MPKI (e.g., KM and SPMV), kernel-initiated cache misses can rapidly exhaust the MSHR entries once the translation bottleneck is alleviated. This resource contention starves the PTW process, preventing NB-WALKER from maintaining its benefits under heavy memory pressure. To mitigate this contention, we introduce a lightweight online profiling and selection mechanism called Adaptive MSHR Reservation (AMR). This mechanism dynamically reserves a portion of the MSHR entries exclusively for PTW-initiated requests, ensuring sustained translation acceleration even under high memory pressure.

The system divides execution time into fixed-length epochs (e.g., 1 K clock cycles). At the end of i-th epoch, the system performs a profiling that samples a 5-bit occupancy counter dedicated to each PTW within the GPC. This counter tracks the total number of PTW requests (denoted as W_i encompassing both pending and in-flight operations). Based on the sampled value W_i, the system categorizes the last epoch into two states: an active state ($W_i > 0$) or an idle state ($W_i = 0$). Driven by this classification, the system determines the optimal number of MSHR entries to be reserved (R_{i+1}) and applies this configuration to the subsequent epoch $i+1$. For a concrete illustration of our mechanism, we assume a representative GPU configuration: each GPC comprises N_{L1} L1 caches, with each cache featuring M_{L1} MSHR entries. Furthermore, the GPC is designed to accommodate a maximum of N_{walk} concurrent PTW requests.

Active State. During the active state, AMR evaluates the estimated MSHR demand per L1 cache (denoted as E_i) by normalizing the aggregate PTW request count (W_i) across the number of L1 caches (N_{L1}) within the GPC, such that $E_i = \lceil W_i / N_{L1} \rceil$. To accommodate PTW request burstiness, the system applies a conservative over-provisioning strategy, incrementing the reservation to $E_i + 1$ during active states. To ensure that PTW metadata does not influence the primary data cache, this reservation is strictly capped by a capacity threshold ($\sigma = \lceil N_{walk} / N_{L1} \rceil$), as shown in Equation 1.

$$R_{i+1} = \max(\min(E_i + 1, \sigma), R_i) \tag{1}$$

TABLE I: Hardware overhead.

Hardware	**Area** (μm^2)	**Power** (mW)
Page walk FSM mux	123.40	0.08
L1 cache MSHR	1099.22	0.90
GPC bidirectional mux	323.79	1.08
Total	**1546.41**	**2.06**

Idle State. For the idle state, we employ a 2-bit saturation counter (C) per GPC to track consecutive idle epochs. At the end of each epoch, the system monitors the MSHR occupancy (N) of a randomly sampled L1 cache to assess the current memory pressure. According to Equation 2, if the PTW remains inactive, R_{i+1} is gradually decreased. Crucially, if the system detects significant resource slack, defined as the occupancy N falling below the capacity threshold ($M_{L1} - \sigma$), it proactively retains a single entry ($R_i = 1$).

$$R_i = \begin{cases} 2, & \text{if } C_i = 1 \\ 1, & \text{if } C_i = 2 \\ 0, & \text{if } C_i \geq 3 \text{ and } N > (M_{L1} - \sigma) \\ 1, & \text{if } C_i \geq 3 \text{ and } N \leq (M_{L1} - \sigma) \end{cases} \tag{2}$$

Our proposed AMR evaluates GPC-private performance counters to determine the number of MSHR entries R_i reserved for NB-WALKER in each L1 cache for the subsequent 1K-cycle epoch. The decision logic requires only seven few-bit comparison operations. Consequently, the value of R_i can be computed within one cycle.

D. Discussion

Hardware Overhead. Given that our baseline GPU is similar to the AMD MI100 GPU (TSMC 7nm process), to evaluate the hardware overhead of NB-WALKER, we implement the proposed modifications in RTL and estimate area and power using Synopsys Design Compiler with the ASAP 7nm Predictive Process Design Kit (PDK) [37]. Power analysis is conducted via gate-level simulation using activity factors extracted from our GPU simulator's performance counters. As summarized in Table I, the integrated components include: (1) augmented FSMs with 3-to-1 MUXes and priority arbitration; (2) L1 MSHR logic with 2-to-1 MUXes and round-robin arbiters; and (3) GPC-level bidirectional interface logic. These components incur area overheads of 123.40 μm^2, 1099.22 μm^2

TABLE II: Synthesis results for the L1 data cache.

Design	Delay @ 1 GHz (ps)	Delay @ 2 GHz (ps)
Baseline (w/o Bypass)	836.34	349.49
Proposed (w/ Bypass)	904.77	349.98
Overhead	**+68.43**	**+0.49**

and 323.79 μm^2, respectively. The aggregate area overhead is 1546.41 μm^2, which represents a negligible 0.0002% of the 750 mm^2 GPU die area. Regarding power, the incremental consumption for the three components is 0.08 mW, 0.90 mW (including PTW-specific MSHR access energy), and 1.08 mW. The total estimated power overhead of 2.06 mW accounts for only 0.0007% of the 300 W TDP of the baseline MI100 GPU. These results demonstrate that NB-WALKER provides significant walk efficiency gains with virtually no impact on the chip's area or power consumption.

Timing Evaluation. To minimize L1 port contention and PTW latency, we add a bypass path via a 2-to-1 MUX between the data array and MSHRs, skipping the tag-matching stage. Furthermore, a 1-bit "PTW" flag is integrated into each MSHR entry to indicate that it is currently servicing a PTW request. To evaluate the impact of these components on the L1 cache's critical path, we perform synthesis on the RTL of the open-source Ventus GPGPU L1 data cache [38]. The synthesis is conducted using Synopsys Design Compiler targeting the ASAP 7 nm Predictive PDK [37] to extract precise timing characteristics. As shown in Table II, synthesis results indicate that under a 1 GHz clock constraint, the bypass logic introduces a combinational delay of 68.43 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.

Compatible with Page Migration. NB-WALKER is fully compatible with standard page migration mechanisms in both heterogeneous and homogeneous multi-GPU systems. Taking a typical CPU-GPU heterogeneous system as an example, the driver ensures memory and translation coherence by flushing GPU caches, aborting in-flight instructions, and invalidating TLBs [35], [39]–[41]. To integrate NB-WALKER, the hardware only needs to drain the PWReqQ, PWRspQ and PWB during this process. Other metadata in the cache MSHRs is automatically cleared by existing hardware flush mechanisms. Consequently, NB-WALKER maintains coherence with negligible overhead and no architectural complexity.

Integration with On-demand Paging. On-demand page placement strategies (e.g., first-touch [42], [43], least-first [44], and LADM [45]) leverage GPU page fault exceptions to automate data movement across PCIe [31], [40]. When a non-resident page is accessed, the GMMU triggers a fault handled by the GPU runtime. Since NB-WALKER utilizes the standard page walk FSM, any invalid PTE encountered inherently triggers a standard page fault entry and raises a page fault exception once the buffer reaches a predefined threshold or a timeout occurs. Thus, NB-WALKER integrates seamlessly into existing fault-handling pipelines with zero additional hardware overhead.

TABLE III: 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 TB/s, 200 ns latency
L1 V-cache	64 KB, 16-way, 32 MSHR, 28 cycles, CU private
L1 S-cache	64 KB, 16-way, 32 MSHR, 28 cycles, shared b/w 4 CUs
L1 I-cache	64 KB, 16-way, 8 MSHR, 28 cycles, shared b/w 4 CUs
L2 cache	8 MB, 16-way, 160 cycles, GPU shared
L1 TLB	32 entries, fully associative, 20 cycles, CU private
L2 TLB	512 entries, 8-way, 40 cycles, GPC shared
L3 TLB	2048 entries, 8-way, 80 cycles, GPU shared
Page table walk	8 page table walkers, GPC shared [15], [17], [22]
Page walk cache	128 entries, 10 cycles, GPU shared [24], [46]

TABLE IV: Benchmark characteristics.

Abbr.	Application	Footprint (MiB)	MPKI L1 $	MPKI L3 TLB
SC	Simple Convolution on Matrix	2048	60.1	0.4
J2D	2-D Jacobi Solver	2048	183.3	2.2
FWT	Fast Warshall Transform	256	145.5	2.3
J1D	1-D Jacobi Solver	2048	211.5	3.2
RED	Reduction Kernel	2048	380.7	6.0
KM	K-Means Clustering	264	335.7	15.6
ST	Stencil Operation on 2-D Matrix	512	189.6	66.2
PR	Page Rank Algorithm	1024	439.5	93.5
MT	Matrix Transpose	512	219.2	191.8
SPMV	Sparse Matrix Vector Mult.	1390	5645.8	377.1
GUPS	Multi-threaded Random Access	256	1071.7	1122.5
GEV	Scalar, Vector and Matrix Multiplication	512	5906.0	5807.6

V. EVALUATION

A. Experimental Setup

1) Baseline Configuration: Our evaluation is conducted using MGPUSim [34], configured with the baseline parameters as summarized in Table III. Following previous work [15],

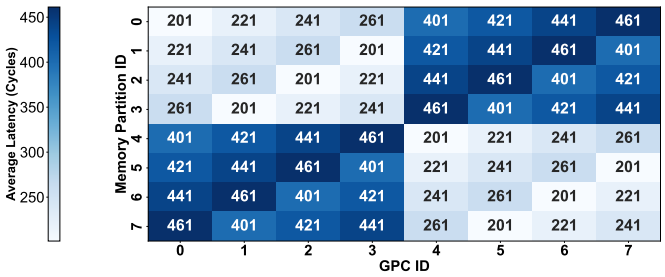


Fig. 11: Non-uniform L2 hit latency in our baseline [8].

[16], [24], [47]–[50], we assume a standard four-level radix-tree page table structure (L4 as root to L1 as leaf) to support a 48-bit virtual address space with 4 KB page granularity [51]. Each page table level consists of 512 entries, where each PTE occupies 8 bytes. In addition, page table pages are interleaved across all DRAM partitions to ensure balanced memory accesses. Additionally, to show the effectiveness of our proposed NB-WALKER, we provide detailed experimental results with large pages in Section V-C. Building upon the architectural assumptions in [16], [19], we augment MGPUSim to model a multi-level TLB hierarchy, incorporating: (1) a GPU-wide shared L3 TLB; (2) a unified page walk cache shared across all GPCs; and (3) dedicated per-GPC page table walkers, each equipped with a private page walk queue to buffer pending translation requests.

Furthermore, to capture NUMA effect within the GPU, we adopt the architectural models and insights proposed by Jin et al. [8]. Specifically, we implement a hierarchical organization comprising 8 GPCs and 8 memory partitions (MPs). Each GPC comprises 16 compute units, while each MP integrates 8 LLC slices and a memory controller. These components are organized into two symmetrical clusters, with 4 GPCs and 4 local MPs per cluster. As shown in Figure 11, an intra-cluster LLC hit (i.e., a GPC requesting data resident in a local MP) incurs a latency of 200–260 cycles. In contrast, inter-cluster requests traversing to a remote MP incur an additional 200-cycle penalty to model the cross-cluster interconnect overhead, resulting in a total remote access latency of 400–460 cycles.

2) Applications: We use 12 applications with diverse memory footprints and address translation pressures, selected from AMD APP SDK [52], Hetero-Mark [53], MGPUSim [34], Pannotia [54], PolyBench [55], and SHOC [56] benchmark suites, as listed in Table IV. To quantify the overhead of address translation, we categorize the applications into two groups based on L3 TLB MPKI: Low-Intensity (L3 TLB MPKI $<$ 10) and High-Intensity (L3 TLB MPKI $\geq$ 10). Since our design targets translation-intensive workloads, applications with low L3 TLB MPKI exhibit lower sensitivity to translation overhead and therefore leave less room for improvement than applications with high L3 TLB MPKI. We report speedup results for all evaluated applications initially for completeness, after which we focus on the high-intensity workloads.

B. Overall Performance

Figure 12 compares three configurations normalized to the baseline GPU: (1) NB-WALKER, which implements the proposed non-blocking address translation architecture; (2) NB-WALKER+AMR, which augments NB-WALKER with Adaptive MSHR Reservation (AMR); and (3) Infinite Walkers, an idealized configuration with unlimited walker resources. Overall, our proposed designs achieve consistent performance gains across all applications with high L3 TLB MPKI, while incurs no performance degradation for applications with low L3 TLB MPKI. The primary performance gains of NB-WALKER stem from its ability to enhance both page-walk and memory-level parallelism. By repurposing underutilized L1 cache MSHRs within the GPC to store metadata for inflight PTW requests, our proposed NB-WALKER effectively decouples the storage resources from the page-walk FSMs. This architectural decoupling mitigates the structural hazards and blocking bottlenecks inherent to conventional, blocking page walkers. On average, NB-WALKER delivers a (harmonic mean) speedup of 2.42× over the baseline for high L3 TLB-MPKI workloads. Furthermore, NB-WALKER+AMR delivers an additional average performance improvement of 1.15× over the NB-WALKER. This gain is primarily attributed to the mitigation of cache MSHR contention. In addition, applications with low L3 TLB MPKI also show a 1.02× average improvement, as NB-WALKER+AMR primarily mitigates translation delays.

To provide an upper-bound reference, we include an idealized configuration equipped with an infinite number of page walkers, offering unlimited page walk parallelism. Notably, NB-WALKER achieves performance speedup near this ideal configuration for the majority of applications. In specific cases, such as ST and PR, our design even outperforms the infinite walker. While the idealized infinite walker configuration provides unlimited parallelism, it handles each page walk as an independent transaction. In contrast, NB-WALKER leverages MSHRs to opportunistically coalesce redundant accesses into a single L2 request. By filtering out the redundant traffic and by satisfying multiple pending walks with a single memory request, our design achieves lower effective latency and higher throughput.

PTW Concurrency. We compare the impact of the evaluated configurations on PTW concurrency using the average number of concurrent PTW requests, as shown in Figure 13. In our baseline GPU configuration, the GPU comprises 8 individual GPCs and the GPU-shared L3 TLB contains 512 MSHR entries, theoretically allowing each GPC's page walker to handle up to 64 concurrent PTW requests. However, our observations reveal that the PTW parallelism in the baseline is severely bottlenecked by the limited capacity of the page walk buffers, achieving only 7.31 concurrent page walks. In contrast, our proposed NB-WALKER significantly enhances PTW parallelism by eliminating the inherent blocking stalls in the baseline walk pipeline. NB-WALKER achieves an average parallelism of 40.68, representing a 5.56× improvement over the baseline. However, for specific benchmarks such as KM and SPMV, NB-WALKER exhibits significantly higher parallelism than both NB-WALKER+AMR and the idealized configuration, yet yields lower overall performance. This counter-intuitive result stems from the high L1 cache MPKI characteristic of these applications. Frequent cache misses exhaust the available L1 MSHRs and prevent NB-WALKER from repurposing MSHR entries. Consequently, NB-WALKER is forced to stall while waiting for free MSHRs, which significantly increases the effective walk latency despite the increased request concurrency.

Page Table Walk Latency. Figure 14 illustrates the normal-

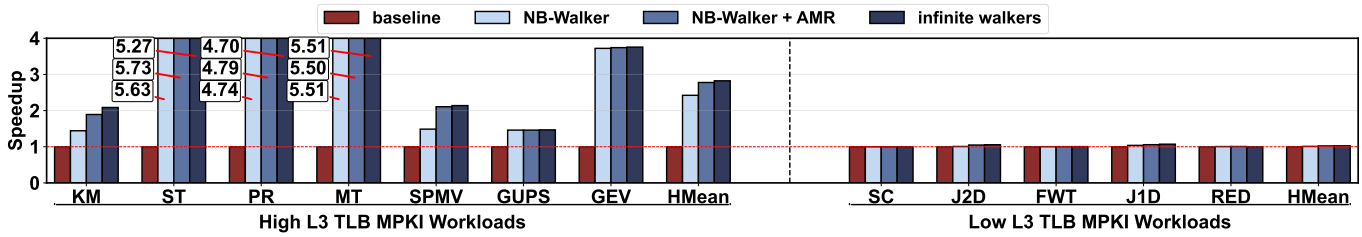


Fig. 12: Speedup of NB-WALKER, NB-WALKER+AMR, and an idealized configuration with infinite walkers.

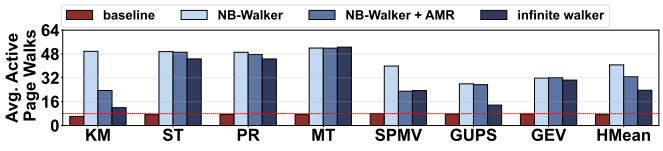


Fig. 13: Average number of concurrent page table walks.

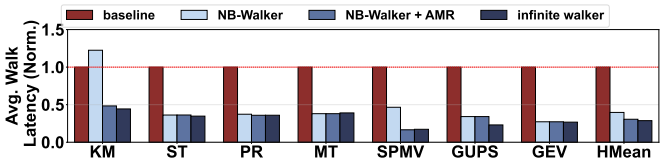


Fig. 14: Normalized page table walk latency.

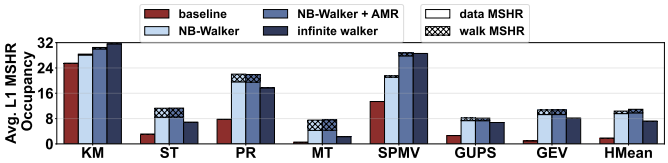


Fig. 15: Breakdown of average L1 MSHR occupancy.

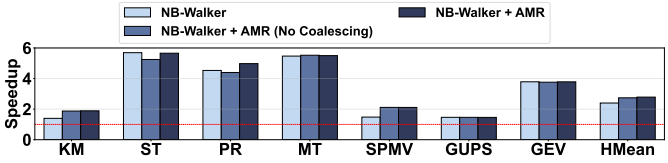


Fig. 16: Speedup over the baseline GPU for NB-WALKER, NB-WALKER+AMR (without L1 MSHR PTE coalescing) and NB-WALKER+AMR.

ized walk latency across all evaluated configurations relative to the baseline. Compared to the baseline, NB-WALKER and NB-WALKER+AMR achieve latency reductions of 2.52× and 3.27×, respectively. However, for the KM workload, NB-WALKER suffers from increased latency due to severe cache miss pressure. Consequently, the lack of available MSHR entries forces PTW requests to queue within the L1 cache. Similarly, SPMV also exhibits significantly higher latency under NB-WALKER compared to NB-WALKER+AMR and infinite walker configuration. Our proposed AMR mechanism effectively addresses this bottleneck by periodically analyzing cache and PTW pressure to dynamically reserve dedicated MSHR entries for NB-WALKER, thereby mitigating resource contention.

L1 MSHR Occupancy. To evaluate the efficacy of our proposed technique for optimizing the memory subsystem, we analyze the occupancy of L1 MSHR entry in various configurations, as illustrated in Figure 15. On average, NB-WALKER and NB-WALKER+AMR enhance L1 MSHR occupancy by a factor of 5.33× and 5.46×, respectively. The increased occupancy reflects the enhanced memory-level parallelism (MLP) facilitated by NB-WALKER, which issues multiple concurrent memory and PTW requests that were previously stalled or serialized. The MSHR entries dedicated to page table walks account for only 3.5% of the total MSHR entries on average.

L1 MSHR Coalescing. Figure 16 shows the speedup of NB-WALKER+AMR over the baseline GPU when L1 MSHR PTE coalescing is disabled. The results indicate that disabling L1 MSHR PTE coalescing on NB-WALKER+AMR has little impact on performance across most applications, reducing the average speedup slightly from 2.78× to 2.74×. This is because L1 MSHR coalescing requires multiple outstanding PTW requests to access the same PTE cacheline. We observe that the average L1 MSHR hit rate for PTWs is only 0.05%, indicating that such coalescing occurs infrequently. Therefore, the speedup of NB-WALKER+AMR mainly comes from non-blocking page table accesses and higher PTW concurrency, rather than from L1 MSHR PTE coalescing.

L2 Cache Traffic. We also evaluate the impact of NB-WALKER on L2 cache traffic. As illustrated in Figure 17, although NB-WALKER significantly increases the memory access concurrency of page table walks, its impact on L2 traffic is almost negligible compared to the baseline GPU, resulting in a marginal increase of only 0.97%.

C. Sensitivity Analyses

Number of GPCs. Figure 18 reports speedup of NB-WALKER+AMR over the baseline as the number of GPCs scales from 1 to 8. NB-WALKER+AMR delivers consistent performance gains across all configurations, indicating that its benefit is robust to different GPC counts. Increasing the number of GPCs raises both translation pressure and aggregate page-walk throughput, since more compute units generate more translation requests and page walkers are distributed

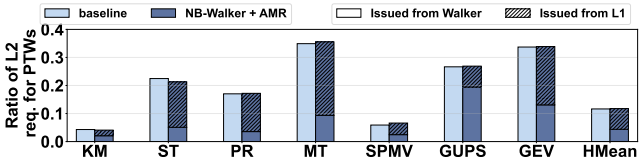


Fig. 17: Fraction of L2 access requests issued for servicing page table walks in the baseline and NB-WALKER+AMR.

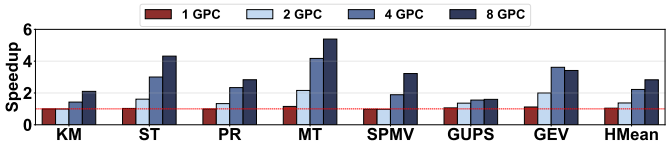


Fig. 18: Speedup of NB-WALKER+AMR over the baseline GPU with varying numbers of GPCs (1, 2, 4 and 8).

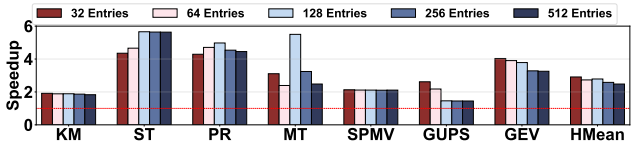


Fig. 19: Speedup of NB-WALKER+AMR over the baseline GPU with varying PWC entries.



Fig. 20: Speedup of NB-WALKER+AMR over the baseline GPU with varying page size (64 KB, 128 KB and 2 MB).

across GPCs. The results show that NB-WALKER+AMR remains effective across this design space by improving intra-GPC resource utilization and alleviating page-walk bottlenecks.

Number of PWC Entries. A larger page walk cache can accommodate more intermediate-level page table entries, thereby reducing memory accesses to upper-level page tables. Figure 19 illustrates the impact of varying PWC capacities, ranging from 32 to 512 entries, on the speedup of NB-WALKER. Although a larger PWC capacity can cover a broader range of upper-level page tables, the memory accesses required by page walkers to fetch leaf page table entries still incur substantial overhead. Consequently, even when a 512-entry PWC caches almost all upper-level page tables (except SPMV), NB-WALKER still achieves an average speedup of 2.48×.

Page Size. While we assume a 4 KB page size thus far, modern GPUs also support larger page sizes, such as 64 KB and 2 MB [57], [58]. To ensure the memory footprint exceeds the L3 TLB coverage, the evaluated workloads feature working set sizes ranging from 4 to 8 GB. As illustrated in Figure 20, when employing 64 KB, 128 KB and 2 MB pages, NB-WALKER+AMR consistently outperforms the baseline GPU, achieving an average speedup of 1.75×, 1.30× and 1.10×, respectively, with peak speedups reaching 5.99×, 3.61× and 2.18×. For KM and SPMV, cache miss pressure significantly outweighs the address translation overhead. Consequently, excessive L1 cache misses emerge as the primary bottleneck when large pages are applied. Conversely, for ST, PR and MT, a large number of TLB misses persist within the GPU even when employing 64 KB pages. For these workloads, NB-WALKER+AMR effectively mitigates the performance bottleneck caused by the insufficient throughput of the page table walkers. Remarkably, even with 2 MB large pages, NB-WALKER+AMR still achieves a 2.18× speedup for MT.

Number of L1 MSHRs. We further investigate the sensitivity of NB-WALKER+AMR's performance to the L1 MSHR capacity by varying the configuration from 16 to 64 entries per L1 cache. As illustrated in Figure 21, the speedup achieved by NB-WALKER+AMR exhibits marginal sensitivity to MSHR scaling. This resilience stems from the fact that the aggregate demand for MSHR entries is effectively distributed across multiple L1 caches within a GPC. Consequently, the MSHR pressure per cache remains moderate. The baseline 32-entry configuration provides sufficient headroom to sustain both memory-level and page table walk parallelism. These results confirm that NB-WALKER attains its performance benefits without requiring over-provisioned MSHR resources.

TLB Hierarchy. Following the configurations established in prior work [15], [16], [24], we evaluate the robustness of NB-WALKER within a two-level TLB hierarchy. In this configuration, we remove the GPC-shared L2 TLBs, such that L1 TLB misses are serviced directly by the GPU-shared Last-Level TLB. This flatter hierarchy significantly amplifies address translation pressure on the remaining structures, demanding higher efficiency from the page-walk subsystem to resolve missed translations. As illustrated in Figure 22, NB-WALKER consistently delivers performance gains comparable to those observed in the baseline three-level TLB architecture. This result demonstrates NB-WALKER 's ability to mitigate translation bottlenecks, regardless of the specific TLB hierarchy.

AMR Profiling Frequency. To evaluate the impact of profiling frequency on performance, we conduct a detailed characterization across frequencies of 250 KHz, 500 KHz and 1 MHz. As shown in Figure 23, compared to the default 1 MHz setting, the 250 KHz and 500 KHz configurations exhibit marginal performance fluctuations of merely 1.51% and 0.34%, respectively, with a maximum slowdown of 13.30% and 3.44% observed for PR. These results indicate that profiling frequency has a negligible impact on most workloads. The slight performance penalty in PR is attributed to the lower sampling rate, which fails to promptly detect the sudden burst of PTW requests generated immediately upon kernel execution, thereby delaying the timely adjustment of the reserved MSHRs.

In addition, Figure 24 shows the time-series PTW/data MSHR occupancy and the AMR-reserved MSHR fraction

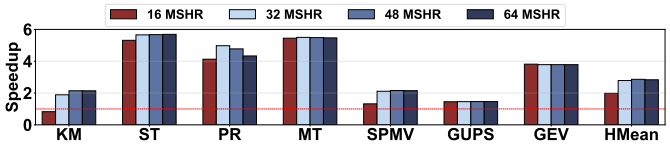


Fig. 21: Speedup of NB-WALKER+AMR over the baseline GPU with varying L1 MSHR capacity (16, 32, 48, 64 entries).

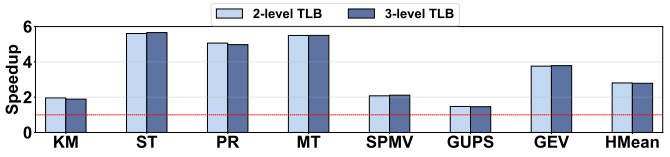


Fig. 22: Speedup of NB-WALKER+AMR over the baseline GPU under two-level and three-level TLB hierarchies.

under different profiling frequencies. For KM where kernel memory requests exert high MSHR pressure, AMR keeps the PTW reservation low while still preserving sufficient headroom for PTWs. In contrast, for MT, the lower kernel MSHR demand allows AMR to reserve more entries for PTWs to sustain page-walk throughput. Across both workloads, AMR does not exhibit persistent oscillation. Additionally, the similar reservation behavior across profiling frequencies further shows that the default 1 MHz frequency is not finely tuned and can be relaxed to lower frequencies without changing AMR's qualitative behavior.

D. Comparison with State-of-the-Art

We further compare NB-WALKER+AMR's performance against two state-of-the-art PTW acceleration paradigms: Marching Page Walks (MPW) [16] and SOFTWALKER [23]. MPW accelerates the PTW process by augmenting the page walk queue to track walk metadata, including page levels and PTE addresses. Specifically, MPW enhances throughput by enabling a single hardware walker to issue batches of memory requests concurrently. In contrast, SOFTWALKER redirects PTW requests to the compute units, employing a dedicated PTW warp (or wavefront) to execute extended ISA instructions. By utilizing the functional units to perform the walk, SOFTWALKER exploits massive thread-level parallelism to handle concurrent translations. As illustrated in Figure 25, while both MPW and SOFTWALKER improve performance over the baseline GPU, NB-WALKER achieves an average speedup of 1.12× (up to 1.80×) and 1.30× (up to 2.25×) over MPW and SOFTWALKER, respectively. Both prior approaches suffer from fundamental synchronization bottlenecks. MPW employs a Walker Status Table to manage batched requests, which implicitly assumes synchronized memory returns. Similarly, SOFTWALKER executes dedicated PTW warps in lockstep, forcing all concurrent translation requests within the same warp to progress synchronously under SIMT execution. However, in modern GPUs characterized by significant NUMA effects, the effective parallelism of the page table walk is

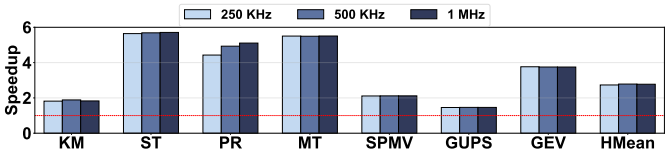
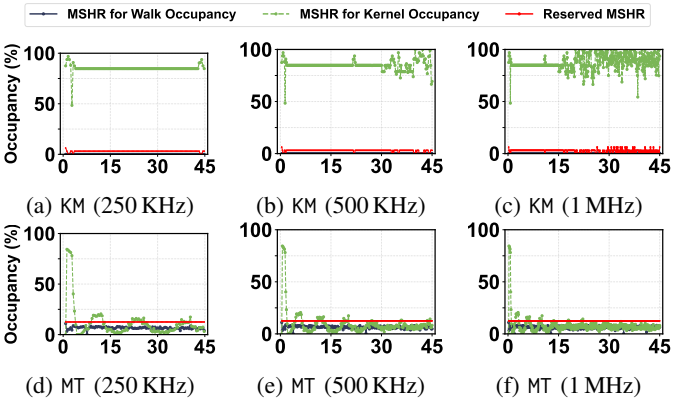


Fig. 23: Speedup of NB-Walker+AMR over the baseline GPU with varying profiling frequency.



(a) KM (250 KHz) (b) KM (500 KHz) (c) KM (1 MHz)

(d) MT (250 KHz) (e) MT (500 KHz) (f) MT (1 MHz)

Fig. 24: Impact of profiling frequency on time-series MSHR utilization and AMR reservations for KM (top row) and MT (bottom row). The x-axis is measured in cycles ($\times 10^4$).

severely throttled because the entire batch or warp is stalled by the slowest memory request. Furthermore, by replacing a dedicated hardware state machine with software execution, SOFTWALKER translates simple arithmetic and memory operations into complex assembly-level instruction sequences, incurring additional latency for each page walk level. Our proposed NB-WALKER addresses these bottlenecks by decoupling page walker storage from the page walk FSM. By leveraging idle L1 cache MSHRs to increase concurrency, NB-WALKER treats each PTW request as an independent entity. Consequently, it eliminates the synchronization overhead induced by NUMA effects and remains robust against irregular memory access patterns and large memory footprints.

E. Comparison to Super-Page Mechanism

To further demonstrate the efficacy of our proposed NB-WALKER, we compare it against the state-of-the-art superpage-based translation mechanisms (e.g., SNAKEBYTE [49]). Following the methodology established in prior work [47], [49], [59], we assume a baseline GPU supporting UVM with parameters as detailed in Table III. This baseline incorporates a contiguity-aware demand paging prefetcher [60] to preserve continuous memory mappings. Specifically, the GPU driver allocates 16 contiguous 4 KB pages (totaling 64 KB) upon a new page fault. To isolate the impact of translation overhead, costs associated with page faults and data migration are excluded from this analysis. SNAKEBYTE reduces translation overhead by recursively coalescing up to eight adjacent pages into larger superpages (up to 8 GB pages),

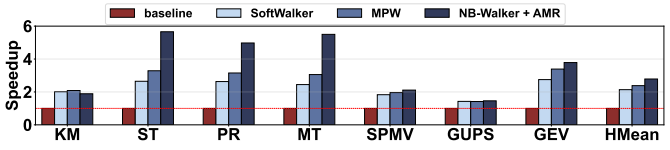


Fig. 25: Performance comparison between NB-WALKER, Marching Page Walks (MPW) [16] and SOFTWALKER [23], normalized to the baseline GPU.

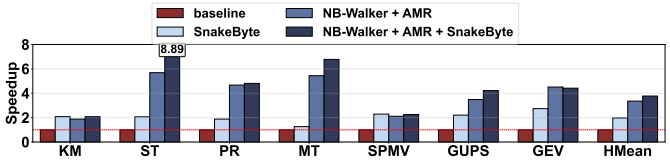


Fig. 26: Performance comparison between NB-WALKER and SNAKEBYTE [49], normalized to the baseline GPU.

effectively increasing TLB reach. Consequently, SNAKEBYTE achieves a 48.97× reduction in L3 TLB MPKI and a 1.97× average performance speedup over the baseline as illustrated Figure 26. However, its benefits are significantly limited in workloads characterized by irregular memory access patterns. In such scenarios, the GPU driver often allocates contiguous physical frames to non-contiguous virtual pages, thereby diminishing the opportunities for superpage coalescing. In contrast, our proposed NB-WALKER mitigates translation overhead by maximizing PTW parallelism. By decoupling the walk process and utilizing idle MSHR resources, NB-WALKER remains robust against fragmented address spaces and irregular access patterns. Results show that NB-WALKER outperforms SNAKEBYTE by 1.70× on average. Furthermore, as NB-WALKER is orthogonal to SNAKEBYTE, we evaluate NB-WALKER on top of SNAKEBYTE, resulting in 1.91× speedup over SNAKEBYTE.

VI. RELATED WORK

Address translation has emerged as a critical performance bottleneck in modern GPU architectures. Prior research has addressed this overhead through two main thrusts: enhancing TLB efficiency and accelerating the PTW process.

TLB Hierarchy Optimizations. A large body of research focuses on increasing the effective capacity of the TLB hierarchy to reduce miss rates. One prominent strategy is resource repurposing, where idle on-chip structures are utilized for address translation. For instance, Kotra et al. [48] proposed harvesting underutilized shared memory and instruction caches to augment TLB storage, while Jaleel et al. [61] introduced a unified cache/TLB architecture (Ducati) that enables the LLC to dynamically allocate entries for both data and translations. Other works have leveraged spatial and temporal locality, such as exploiting page sharing behaviors [12], [19], [62]–[64] or employing variable-sized pages to expand TLB reach [47], [49], [59], [65]. While these techniques effectively filter many translation requests, they do not alleviate the limited walk throughput incurred when a TLB miss inevitably occurs.

Accelerating Page Table Walks. To mitigate TLB miss penalties, researchers have sought to improve PTW throughput and efficiency. One way optimizes scheduling, such as MASK [66], which introduces a specialized page walk cache and prioritizes translation-related memory requests. To further reduce compute unit stalls, Shin et al. [17] proposed reordering PTW requests based on their impact on execution, while subsequent work [15] exploited PTE locality to serve multiple requests with a single cache line fetch. To further increase MLP, Lee et al. [16] introduced Marching Page Walks, enabling a single walker to issue concurrent requests. Recent explorations even involve alternative compute paradigms, such as SoftWalker [23] and cuPTW [25], which offload PTW tasks to idle compute units, or speculative approaches like Avatar [67]. In the context of Multi-Chiplet Module (MCM) GPUs [68], Barre Chord [69] optimizes inter-chiplet translation through intelligent page allocation. Despite these advancements, existing PTW acceleration methods often necessitate non-trivial hardware modifications or dedicated storage structures. Furthermore, most prior work ignores the NUMA effect existing in modern monolithic GPUs. In contrast, our proposed mechanism, NB-WALKER, identifies a unique opportunity in the underutilized L1 MSHR entries. By repurposing these resources to manage pending translation requests, NB-WALKER significantly boosts address translation throughput with near-zero area overhead, providing a more hardware-efficient solution than prior architectural extensions.

VII. CONCLUSION

The NUMA effect in modern GPU systems significantly exacerbates the address translation bottleneck. To mitigate this, we propose NB-WALKER, a novel architecture that repurposes idle L1 cache MSHRs to accelerate page table walks. By buffering in-flight PTW metadata within these MSHRs, NB-WALKER transforms the traditionally blocking page table walking process into non-blocking execution, thereby substantially boosting translation throughput. Our evaluation demonstrates that NB-WALKER+AMR achieves a speedup of 2.42× to 2.78× over the baseline GPU architecture by significantly enhancing PTW efficiency.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments. This work was supported in part by the National Natural Science Foundation of China (No. 62402411) and the Guangdong Provincial Project (No. 2023QN10X252). This research was conducted on the High-Performance Computing Platform of HKUST(GZ).

REFERENCES

[1] Y. Dai, Y. Zhang, and X. Tang, "CEGMA: Coordinated elastic graph matching acceleration for graph matching networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 584–597. [Online]. Available: https://ieeexplore.ieee.org/document/10070956/

[2] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the GPU — past, present and future," *Medical Image Analysis*, vol. 17, no. 8, pp. 1073–1094, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361841513000820

[3] S. Feng, S. Pal, Y. Yang, and R. G. Dreslinski, "Parallelism analysis of prominent desktop applications: An 18- year perspective," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 202–211. [Online]. Available: https://ieeexplore.ieee.org/document/8695665/

[4] T. Ge, T. Zhang, and H. Liu, "ngAP: Non-blocking Large-scale Automata Processing on GPUs," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. La Jolla CA USA: ACM, Apr. 2024, pp. 268–285. [Online]. Available: https://dl.acm.org/doi/10.1145/3617232.3624848

[5] T. Ge, X. Chu, and H. Liu, "Interleaved bitstream execution for multi-pattern regex matching on GPUs," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 385–400. [Online]. Available: https://doi.org/10.1145/3725843.3756052

[6] Z. Wu, H. Zhao, H. Liu, W. Wen, and J. Li, "gHyPart: GPU-friendly end-to-end hypergraph partitioner," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 1, Mar. 2025. [Online]. Available: https://doi.org/10.1145/3711925

[7] B. Zeng, Z. Wu, H. Liu, and X. Chen, "DSTree: Data-driven synchronous traversals for decision forests on GPUs," in *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2026, pp. 401–415. [Online]. Available: https://ieeexplore.ieee.org/document/11575326

[8] Z. Jin, C. Rocca, J. Kim, H. Kasan, M. Rhu, A. Bakhoda, T. M. Aamodt, and J. Kim, "Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Austin, TX, USA: IEEE, Nov. 2024, pp. 885–898. [Online]. Available: https://ieeexplore.ieee.org/document/10764573/

[9] A. Tee, N. Curtis, N. Wolfe, and D. Wong, "The MALL is Open: Exploring Shared Caches and Latency in AMD CDNA™ 3 GPUs," in *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St Louis MO USA: ACM, Nov. 2025, pp. 1110–1116. [Online]. Available: https://dl.acm.org/doi/10.1145/3731599.3767487

[10] J. Wahlgren, G. Schieffer, R. Shi, E. A. León, R. Pearce, M. Gokhale, and I. Peng, "Dissecting CPU-GPU Unified Physical Memory on AMD MI300A APUs," Aug. 2025, arXiv:2508.12743. [Online]. Available: http://arxiv.org/abs/2508.12743

[11] P. Rogers, "Heterogeneous system architecture overview," in *2013 IEEE Hot Chips 25 Symposium (HCS)*, 2013, pp. 1–41. [Online]. Available: https://ieeexplore.ieee.org/document/7478286

[12] B. Li, J. Yin, Y. Zhang, and X. Tang, "Improving Address Translation in Multi-GPUs via Sharing and Spilling aware TLB Design," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Virtual Event Greece: ACM, Oct. 2021, pp. 1154–1168. [Online]. Available: https://dl.acm.org/doi/10.1145/3466752.3480083

[13] B. Li, J. Yin, A. Holey, Y. Zhang, J. Yang, and X. Tang, "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)*. Montreal, QC, Canada: IEEE, Feb. 2023, pp. 456–470. [Online]. Available: https://ieeexplore.ieee.org/document/10071054/

[14] B. Li, Y. Guo, Y. Wang, A. Jaleel, J. Yang, and X. Tang, "IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*. Toronto ON Canada: ACM, Oct. 2023, pp. 1163–1177. [Online]. Available: https://dl.acm.org/doi/10.1145/3613424.3614269

[15] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular GPU applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 352–363. [Online]. Available: https://ieeexplore.ieee.org/document/8574553

[16] J. Lee, G. Ko, M. K. Yoon, I. Jeong, Y. Oh, and W. W. Ro, "Marching Page Walks: Batching and Concurrent Page Table Walks for Enhancing GPU Throughput," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2025, pp. 1662–1677. [Online]. Available: https://ieeexplore.ieee.org/document/10946291/

[17] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular GPU applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 180–192. [Online]. Available: https://ieeexplore.ieee.org/document/8416827

[18] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, and M. Guo, "Oversubscribing GPU unified virtual memory: Implications and suggestions," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 67–75. [Online]. Available: https://doi.org/10.1145/3489525.3511691

[19] B. Li, Y. Wang, T. Wang, L. Eeckhout, J. Yang, A. Jaleel, and X. Tang, "STAR: Sub-Entry Sharing-Aware TLB for Multi-Instance GPU," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2024, pp. 309–323. [Online]. Available: https://ieeexplore.ieee.org/document/10764582/

[20] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, "Tunnels for bootlegging: Fully reverse-engineering GPU TLBs for challenging isolation guarantees of NVIDIA MIG," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 960–974. [Online]. Available: https://doi.org/10.1145/3576915.3616672

[21] S. E. Molnar, J. K. Gupta, J. L. Deming, S. H. Duncan, and J. Smith, "Method and system for implementing multi-stage translation of virtual addresses," Oct. 2018, US Patent 10,114,760 B2. [Online]. Available: https://patents.google.com/patent/US10114760B2/en

[22] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 161–171. [Online]. Available: https://ieeexplore.ieee.org/document/7482091

[23] S. Jang, J. Park, Y. Lee, O. Kwon, D. Kim, J. Seok, and S. Hong, "SoftWalker: Supporting Software Page Table Walk for Irregular GPU Applications," in *Proceedings of the 2025 58th IEEE/ACM International Symposium on Microarchitecture*. Seoul Republic of Korea: ACM, Oct. 2025, pp. 401–417. [Online]. Available: https://dl.acm.org/doi/10.1145/3725843.3756056

[24] B. Pratheek, N. Jawalkar, and A. Basu, "Improving GPU multi-tenancy with page walk stealing," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 626–639. [Online]. Available: https://ieeexplore.ieee.org/document/9407125

[25] Z. Chen, T. Ge, L. Eeckhout, H. Liu, and J. Huang, "cuPTW: Leveraging idle compute units for massively parallel GPU page table walks," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 10, no. 2, May 2026. [Online]. Available: https://doi.org/10.1145/3805633

[26] NVIDIA Corporation, *CUDA C++ Programming Guide*, NVIDIA Corporation, 2025, accessed: 2025-10-14. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

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

[28] H. Abdelkhalik, Y. Arafa, N. Santhi, and A.-H. A. Badawy, "Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2022, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9926299/

[29] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, "Benchmarking and Dissecting the Nvidia Hopper GPU Architecture," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2024, pp. 656–667. [Online]. Available: https://ieeexplore.ieee.org/document/10579250/

[30] A. Jarmusch, N. Graddon, and S. Chandrasekaran, "Dissecting the NVIDIA Blackwell Architecture with Microbenchmarks," Jul. 2025, arXiv:2507.10789. [Online]. Available: http://arxiv.org/abs/2507.10789

[31] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-Aware Unified Memory Management in GPUs for Irregular Workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating*

Systems. Lausanne Switzerland: ACM, Mar. 2020, pp. 1357–1370. [Online]. Available: https://dl.acm.org/doi/10.1145/3373376.3378529

[32] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, 1981, pp. 81–87. [Online]. Available: https://dl.acm.org/doi/10.5555/800052.801868

[33] K. I. Farkas and N. P. Jouppi, "Complexity/performance tradeoffs with non-blocking loads," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 211–222, Apr. 1994. [Online]. Available: https://doi.org/10.1145/192007.192029

[34] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "MGPUSim: enabling multi-GPU performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*. Phoenix Arizona: ACM, Jun. 2019, pp. 197–209. [Online]. Available: https://dl.acm.org/doi/10.1145/3307650.3322230

[35] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 568–578. [Online]. Available: https://ieeexplore.ieee.org/document/6835965/

[36] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006. [Online]. Available: https://ieeexplore.ieee.org/document/1599540/

[37] V. Vashishtha, M. Vangala, and L. T. Clark, "ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 992–998. [Online]. Available: https://ieeexplore.ieee.org/document/8203889

[38] J. Li, K. Yang, C. Jin, X. Liu, Z. Yang, F. Yu, Y. Shi, M. Ma, L. Kong, J. Zhou, H. Wu, and H. He, "Ventus: A high-performance open-source GPGPU based on RISC-V and its vector extension," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 276–279. [Online]. Available: https://ieeexplore.ieee.org/document/10818098

[39] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 354–365. [Online]. Available: https://ieeexplore.ieee.org/document/7056046/

[40] T. Baruah, Y. Sun, A. T. Dincer, S. A. Mojumder, J. L. Abellan, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. San Diego, CA, USA: IEEE, Feb. 2020, pp. 596–609. [Online]. Available: https://ieeexplore.ieee.org/document/9065453/

[41] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for GPUs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Barcelona, Spain: IEEE, Mar. 2016, pp. 345–357. [Online]. Available: http://ieeexplore.ieee.org/document/7446077/

[42] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: NUMA-aware GPUs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge Massachusetts: ACM, Oct. 2017, pp. 123–135. [Online]. Available: https://dl.acm.org/doi/10.1145/3123939.3124534

[43] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka: IEEE, Oct. 2018, pp. 339–351. [Online]. Available: https://ieeexplore.ieee.org/document/8574552/

[44] X. Zhao, M. Jahre, Y. Tang, G. Zhang, and L. Eeckhout, "NUBA: Non-Uniform Bandwidth GPUs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Vancouver BC Canada: ACM, Jan. 2023, pp. 544–559. [Online]. Available: https://dl.acm.org/doi/10.1145/3575693.3575745

[45] M. Khairy, V. Nikiforov, D. Nellans, and T. G. Rogers, "Locality-Centric Data and Threadblock Management for Massive GPUs," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Athens, Greece: IEEE, Oct. 2020, pp. 1022–1036. [Online]. Available: https://ieeexplore.ieee.org/document/9251964/

[46] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48–59, Jun. 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1816038.1815970

[47] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a GPU memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge Massachusetts: ACM, Oct. 2017, pp. 136–150. [Online]. Available: https://dl.acm.org/doi/10.1145/3123939.3123975

[48] J. B. Kotra, M. LeBeane, M. T. Kandemir, and G. H. Loh, "Increasing GPU Translation Reach by Leveraging Under-Utilized On-Chip Resources," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Virtual Event Greece: ACM, Oct. 2021, pp. 1169–1181. [Online]. Available: https://dl.acm.org/doi/10.1145/3466752.3480105

[49] J. Lee, J. M. Lee, Y. Oh, W. J. Song, and W. W. Ro, "SnakeByte: A TLB Design with Adaptive and Recursive Page Merging in GPUs," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2023, pp. 1195–1207. [Online]. Available: https://ieeexplore.ieee.org/document/10071063/?arnumber=10071063

[50] B. Pratheek, N. Jawalkar, and A. Basu, "Designing Virtual Memory System of MCM GPUs," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Chicago, IL, USA: IEEE, Oct. 2022, pp. 404–422. [Online]. Available: https://ieeexplore.ieee.org/document/9923857/

[51] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 743–758, Apr. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2654822.2541942

[52] AMD, "Tools and SDKs," 2021. [Online]. Available: https://developer.amd.com/amdaccelerated-parallel-processing-app-sdk/

[53] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/document/7581262

[54] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 185–195. [Online]. Available: https://ieeexplore.ieee.org/document/6704684

[55] L.-N. Pouchet and T. Yuki, "Polybench," 2010. [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[56] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010, pp. 63–74. [Online]. Available: https://doi.org/10.1145/1735688.1735702

[57] NVIDIA Corporation, "NVIDIA Pascal MMU," 2019, accessed: 2026-06-05. [Online]. Available: https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf

[58] Advanced Micro Devices, Inc., "Radeon® Pro Software Enterprise Edition 17.Q4 for Linux® Release Notes," Oct. 2017, accessed: 2026-06-05. [Online]. Available: https://www.amd.com/en/resources/support-articles/release-notes/RN-PRORAD-LIN-AMDGPUPRO-17-40.html

[59] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 258–269. [Online]. Available: https://ieeexplore.ieee.org/document/6493625

[60] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and Exploiting Contiguity for Fast Memory Virtualization," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020,

pp. 515–528. [Online]. Available: https://ieeexplore.ieee.org/document/9138910/

[61] A. Jaleel, E. Ebrahimi, and S. Duncan, "DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 1, pp. 1–24, Mar. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3309710

[62] T. Baruah, Y. Sun, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Virtual Event GA USA: ACM, Sep. 2020, pp. 455–466. [Online]. Available: https://dl.acm.org/doi/10.1145/3410463.3414639

[63] Y. Du, M. Liu, Y. Yang, M. Zhang, and X. Tang, "Enhancing GPU Performance via Neighboring Directory Table Based Inter-TLB Sharing," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, Oct. 2022, pp. 146–153. [Online]. Available: https://ieeexplore.ieee.org/document/9978340/

[64] B. Li, Y. Wang, and X. Tang, "Orchestrated Scheduling and Partitioning for Improved Address Translation in GPUs," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, Jul. 2023, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10247943/

[65] K. Gosakan, J. Han, W. Kuszmaul, I. N. Mubarek, N. Mukherjee, K. Sriram, G. Tagliavini, E. West, M. A. Bender, A. Bhattacharjee, A. Conway, M. Farach-Colton, J. Gandhi, R. Johnson, S. Kannan, and D. E. Porter, "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*. Vancouver BC Canada: ACM, Mar. 2023, pp. 433–448. [Online]. Available: https://dl.acm.org/doi/10.1145/3582016.3582021

[66] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 503–518, Nov. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3296957.3173169

[67] J. Park, O. Kwon, Y. Lee, S. Kim, G. Byeon, J. Yoon, P. J. Nair, and S. Hong, "A Case for Speculative Address Translation with Rapid Validation for GPUs," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2024, pp. 278–292. [Online]. Available: https://ieeexplore.ieee.org/document/10764498/?arnumber=10764498

[68] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, Jun. 2017, pp. 320–332. [Online]. Available: https://dl.acm.org/doi/10.1145/3079856.3080231

[69] Y. Feng, S. Na, H. Kim, and H. Jeon, "Barre Chord: Efficient Virtual Memory Translation for Multi-Chip-Module GPUs," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2024, pp. 834–847. [Online]. Available: https://ieeexplore.ieee.org/stampPDF/getPDF.jsp?tp=&arnumber=10609639&ref=&tag=1

ARTIFACT APPENDIX

A. Abstract

Our artifact provides a simulator source code archive and scripts for compiling the simulator, running the experiments, and parsing the resulting metrics. We also provide scripts to reproduce the results shown in Figure 12, Figure 13, Figure 14, Figure 25, and Figure 26.

B. Artifact Check-list (Meta-information)

- **Compilation:** Go
- **Run-time environment:** Tested on Linux and macOS
- **Hardware:** Tested on both x86 and Arm platforms
- **Metrics:** Speedup, number of active walkers, and page-walk latency
- **Disk space required (approximately):** 2 GB
- **Time required to prepare the workflow (approximately):** One (1) hour
- **Time required to complete the experiments (approximately):** Forty-eight (48) hours when run in parallel
- **Publicly available:** Yes
- **Archived (provide DOI):** 10.5281/zenodo.21488604

C. Description

We evaluate NUMA GPUs under a range of configurations using an extended version of MGPUSim. Our evaluation includes 12 benchmarks, comprising five low-L3 TLB MPKI applications and seven high-L3 TLB MPKI applications. The evaluated configurations are organized into two groups.

Configurations. The first group compares different page table walker designs in a NUMA GPU system:

- **Baseline**: A baseline NUMA GPU equipped with blocking page table walkers.
- **NBWalker**: Our proposed non-blocking page table walker design.
- **NBWalker+AMR**: NB-WALKER augmented with Adaptive MSHR Reservation.
- **Infinite-Walker**: An idealized configuration with unlimited page table walker resources.
- **MPW**: Marching Page Walks.
- **SoftWalker**: SoftWalker.

The second group evaluates our design in a UVM setting and compares it with SNAKEBYTE:

- **UVM**: A baseline GPU employing UVM and a contiguity-aware demand-paging prefetcher.
- **UVM+NBWalker+AMR**: The UVM baseline augmented with NB-WALKER+AMR.
- **UVM+SnakeByte**: The UVM baseline augmented with SNAKEBYTE.
- **UVM+SnakeByte+NBWalker+AMR**: The SNAKEBYTE configuration further augmented with NB-WALKER+AMR.

Evaluation. The primary results of our evaluation are presented in Figure 12, Figure 13, Figure 14, Figure 25, and Figure 26.

- Figure 12 reports the speedup achieved by NB-WALKER, NB-WALKER+AMR, and an idealized configuration with an unlimited number of page table walkers.
- Figure 13 reports the average number of concurrent page table walks.
- Figure 14 reports the normalized page table walk latency.
- Figure 25 compares the performance of NB-WALKER, MPW, and SOFTWALKER, normalized to that of the baseline GPU.
- Figure 26 compares the performance of NB-WALKER and SNAKEBYTE under the UVM configuration, normalized to that of the baseline GPU.

1) How to Access the Artifact: Please download the artifact archive in `tar.gz` format from the following URL:
https://doi.org/10.5281/zenodo.21488604

2) Hardware Dependencies: We ran our simulations on x86-64 machines equipped with Intel or AMD processors, as well as on an Arm-based Apple M1 Pro machine. We expect the simulations to produce equivalent results when compiled and executed on other architectures.

- A single experimental run for one configuration requires approximately 400–500 GB of memory.
- Each run requires 12 CPU cores, with one core allocated to each of the 12 workloads.

3) Software Dependencies: MGPUSim is implemented in Go. Therefore, the artifact requires a `Go` development environment, version 1.23.4 or later. In addition, `Python3`, together with the `pandas`, `numpy`, and `matplotlib` packages, is required for processing the experimental data.

D. Installation

Download and extract the artifact archive in `tar.gz` format. Then, change to the `NBWalker-MICRO2026` directory:

```
cd NBWalker-MICRO2026
```

The `scripts` directory contains an environment-setup script, compilation scripts for all evaluated configurations, scripts for executing the experiments, and scripts for reproducing experimental results reported in the paper.

Run the following scripts in order to set up and generate the workload executables:

```
source scripts/0_setup.sh
./scripts/1_generate.sh
```

These commands create a `runs` directory under the root directory, i.e., `NBWalker-MICRO2026`. The directory contains ten sub-directories corresponding to the configurations evaluated in this work. Each configuration directory contains multiple benchmark-specific sub-directories, each of which includes the corresponding benchmark binary and an execution script. For example, `runs/baseline/kmeans/kmeans.sh` runs the `kmeans` benchmark under the `baseline` configuration.

Additional scripts are provided to execute the workloads in batches, as described in the following section.

E. Experiment Workflow

We provide the script `scripts/2_run_simulation.py` to execute the experiments. The `---configuration` argument specifies the configuration to evaluate. The script then launches all experiments associated with the selected configuration in batch mode.

For example, to run the experiments for the `Baseline` configuration, please run:

```
./scripts/2_run_simulation.py \
    ---configuration Baseline
```

Experiments for each configuration typically require 400–500 GB of peak memory and 12 CPU cores, with one CPU core allocated to each workload. The script launches the experiments as background processes. Most experiments complete within 24 hours; however, `SPMV`, `KM`, and `ST` may take up to 48 hours. We recommend monitoring the running processes and their resource usage using `top`.

After all experiments for a configuration have completed, run `scripts/3_collect_csv.py` to collect the simulator outputs. The target configuration must be specified using the `---configuration` argument. The script creates a sub-directory under `results` folder containing the results of all experiments associated with the specified configuration.

For example, to collect the experimental results for the `Baseline` configuration, please run:

```
./scripts/3_collect_csv.py \
    ---configuration Baseline
```

Finally, we provide the following scripts to reproduce the experimental results reported in the paper:

- `scripts/4_output_figure_12.py`
- `scripts/4_output_figure_13.py`
- `scripts/4_output_figure_14.py`
- `scripts/4_output_figure_25.py`
- `scripts/4_output_figure_26.py`

Each script automatically parses the experimental data for the relevant configurations and writes the processed results, such as normalized kernel execution times, to the corresponding `.csv` and `.png` files in the `reproduce` directory.

For example, to reproduce the results shown in Figure 12, please run:

```
python3 ./scripts/4_output_figure_12.py
```

F. Evaluation and Expected Results

The `.csv` files generated in the `reproduce` directory contain the processed data underlying the corresponding figures presented in the paper. The reproduced results are expected to be consistent with the values reported in those figures. In addition, the scripts generate the corresponding figures in `.png` format and save them in the same directory.

G. Notes

Please ensure that all scripts have executable permissions. If a script is not executable, use `chmod` to set its executable bit:

```
chmod +x {scriptname.ext}
```