SoftWalker: Supporting Software Page Table Walk for Irregular GPU Applications

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

Junhyeok Park*
Electronics and Telecommunications Research Institute
Daejeon, Republic of Korea
vzx00770@etri.re.kr

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

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

Donghyun Kim
Sungkyunkwan University
Suwon, Republic of Korea
ehdgus583205@skku.edu

Juyoung Seok
Sungkyunkwan University
Suwon, Republic of Korea
tjrwndud5@skku.edu

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

Abstract

Address translation has become a significant and growing performance bottleneck in modern GPUs, especially for emerging irregular applications with high TLB miss rates. The limited concurrency of hardware Page Table Walkers (PTWs), due to their small and fixed number, causes severe contention and substantial queueing delays under high translation pressure, which significantly degrades performance.

This paper introduces *SoftWalker*, a novel, scalable, and flexible framework that fundamentally shifts the GPU page table walking from fixed-function hardware to software execution. *SoftWalker* leverages the GPU's massive thread-level parallelism by dynamically dispatching specialized, lightweight software threads running on GPU cores to handle TLB misses requiring page table walks. In addition, to expand L2 TLB MSHR capacity on demand, *SoftWalker* incorporates In-TLB MSHRs, a key innovation that repurposes underutilized L2 TLB entries to track outstanding misses when existing MSHRs are saturated. By alleviating MSHR-induced contention, this design preserves the key advantage of highly parallel page table walking in software. *SoftWalker* enables thousands of concurrent page table walks, significantly reducing PTW-level contention and translation queueing delays. As a result, it achieves an average reduction of 72.8% in page walk latency and delivers an average speedup of 2.24× (3.94× for irregular workloads).

CCS Concepts

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

*This work was done while the author was at Sungkyunkwan University.

Keywords

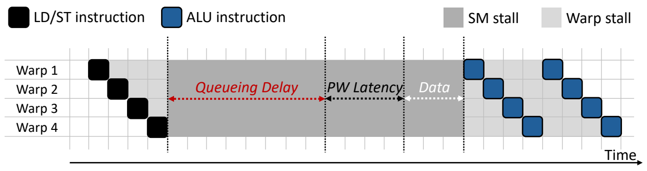
GPGPU, Virtual Memory, Address Translation, Page Table Walk

ACM Reference Format:
Sungbin Jang, Junhyeok Park, Yongho Lee, Osang Kwon, Donghyun Kim, Juyoung Seok, and Seokin Hong. 2025. SoftWalker: Supporting Software Page Table Walk for Irregular GPU Applications. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3725843.3756056

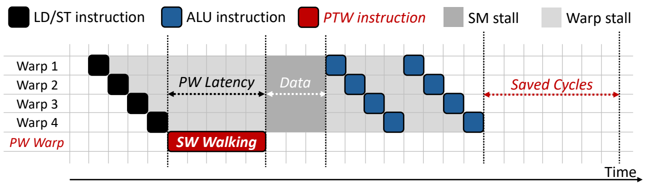
1 Introduction

Modern Graphics Processing Units (GPUs) have become indispensable accelerators for a wide range of tasks, from scientific simulations and data analytics to deep learning and graphics rendering [11, 26, 40, 64, 67, 92, 95]. To support the increasingly large datasets and complex memory access patterns characteristic of these applications, GPUs have incorporated sophisticated virtual memory systems. This trend is further accelerated by the advent of high-speed interconnects like NVLink [63, 69] and Compute Express Link (CXL) [15, 51, 73]. These technologies allow multiple GPUs or system-level memory expansions to be combined into a single, large memory domain, positioning the virtual memory system as the cornerstone for abstracting data location and enabling seamless access across the expanded, heterogeneous memory space. Consequently, these virtual memory systems provide benefits like larger address spaces, programmability, and support for advanced features such as unified memory and demand paging [3, 43, 84]. Central to GPU virtual memory is the address translation mechanism, which maps virtual addresses generated by GPU threads to physical addresses. This process typically relies on a hierarchy of Translation Lookaside Buffers (TLBs) to cache recent translations and hardware Page Table Walkers (PTWs) to resolve TLB misses by traversing multi-level page tables residing in memory [36, 46, 47, 60, 75, 76].

Despite many benefits, address translation is increasingly becoming a significant bottleneck, particularly for applications with



(a) Baseline



(b) *SoftWalker*

Figure 1: *SoftWalker* Overview

irregular memory access patterns [6, 7, 35, 45, 49, 72, 85, 86]. Unlike regular workloads with coalesced memory accesses, irregular applications (common in graph processing, sparse linear algebra, and database operations) often exhibit scattered, fine-grained memory accesses with poor spatial and temporal locality. Consequently, threads within the same warp often access different memory pages, exceeding the limited capacity of on-chip TLBs and resulting in frequent TLB misses.

Conventional GPUs attempt to handle these TLB misses using a small, fixed number of hardware-based PTWs [6, 23, 45, 49, 52, 53, 72, 77, 78, 86, 94]. While this design is sufficient for workloads with high TLB hit rates, its limited scalability in parallelizing page walks through hardware PTWs introduces a critical bottleneck under high translation pressure. When numerous TLB misses occur simultaneously across thousands of active threads, the limited number of hardware PTWs becomes a heavily contended resource. This contention leads to substantial queueing delays, as translation requests are buffered until a PTW becomes available. Our experimental results reveal that for irregular workloads, queueing delay accounts for 95% of the total page table walk latency on average (see Section 3.3 for more details), far exceeding the actual page table traversal time. While scaling the number of hardware PTWs may reduce queueing delays, it incurs substantial area and power overheads and requires proportionally larger associated structures like Miss Status Holding Registers (MSHRs) and Page Walk Buffers (PWBs), posing scalability challenges [23, 77, 86].

To address this challenge, this paper introduces ***SoftWalker***, a scalable and flexible framework that performs GPU page table walking entirely in software. Instead of relying solely on fixed-function hardware, *SoftWalker* leverages the inherent massive Thread-Level Parallelism (TLP) of the GPU itself to perform page table walks. The key idea is to dynamically launch specialized, lightweight GPU threads (termed ***Page Walk Warps*** or ***PW Warps***) to handle TLB misses by leveraging idle GPU cycles. Figure 1 depicts the conceptual overview of *SoftWalker*. When there are a massive number of TLB misses in the baseline architecture, the Streaming Multiprocessors (SMs)[1] are stalled due to high queueing delay caused by contention in a limited number of PTWs. In contrast, our approach offloads the TLB misses to one of the SMs by launching a PW Warp. This warp executes a lightweight software routine that traverses the page table hierarchy, thereby repurposing idle GPU execution resources to serve as on-demand page table walkers. Since the GPU can launch enough PW Warps to handle a large number of concurrent TLB misses, *SoftWalker* can eliminate queueing delays caused by limited PTW resources. As a result, our approach resolves stalls associated with address translation faster than the baseline, leading to significant performance improvement.

[1]We use NVIDIA’s terminology in this paper. However, our proposed technique is general and applicable to other GPU architectures (e.g., AMD, Intel).

SoftWalker incorporates two architectural innovations to enable this software-based approach by overcoming associated challenges. First, it introduces PW Warps, which are isolated from regular user warps to maintain security and avoid interfering with application parallelism. *SoftWalker* implements PW Warps on each SM with minimal hardware overhead. Second, to address the bottleneck caused by the limited number of L2 TLB MSHRs, it introduces ***In-TLB MSHRs*** inspired by In-Cache MSHRs [21]. This mechanism repurposes underutilized L2 TLB entries themselves to temporarily store metadata for pending TLB misses when the dedicated hardware MSHRs are full. In-TLB MSHR notably expands the system’s capacity to track concurrent outstanding translations without requiring large, power-hungry hardware MSHR structures and allows more page walks to be processed in parallel.

Together, *SoftWalker* can support thousands of concurrent page walks, effectively eliminating the contention-induced queueing delays that plague hardware-based page walk systems under high TLB miss pressure. Converting stall cycles, often prevalent during long-latency memory operations (including address translation) in irregular workloads, into productive page-walking work, *SoftWalker* improves address translation throughput and overall GPU resource utilization. Our evaluation demonstrates that *SoftWalker* significantly reduces page table walk latency by an average of 72.6% and achieves substantial application speedups of 2.24x on average across diverse workloads (3.94x for irregular workloads). To ensure compatibility with latency-sensitive regular workloads, *SoftWalker* can operate in a hybrid mode that retains conventional hardware PTWs and selectively uses software-based walkers only when necessary, enabling flexible and practical deployment across diverse GPU environments.

The key contributions of this paper are as follows:

- **Detailed Analysis on Page Walk Contentions.** We provide a detailed analysis of GPU workloads that show irregular memory access patterns, highlighting severe contention in page table walks. This motivates the need for scalable page walk mechanisms.
- **Software-Defined Page Walk Framework.** We propose *SoftWalker*, the first software-defined framework for GPU page table walk. It utilizes specialized software threads (*PW Warps*) executed on SMs to dynamically scale page walk concurrency by leveraging the GPU’s inherent thread-level parallelism and converting idle compute resources into on-demand walkers. This design addresses the scalability limitations of fixed hardware PTWs fundamentally.
- **Architectural Integration.** We detail the architectural support for *SoftWalker*, including PW Warp management (*Soft-PWB, SoftWalker Controller, Request Distributor*) and minimal

ISA extensions (LDPT, FL2T, FPWC, FFB). Furthermore, we introduce *In-TLB MSHRs*, a lightweight and flexible technique that repurposes underutilized L2 TLB entries as temporary storage to track outstanding TLB misses.

- **Evaluation.** We evaluate *SoftWalker* across diverse workloads and various configurations, demonstrating on average 72.8% reduction in translation latency and a 2.24× performance improvement.

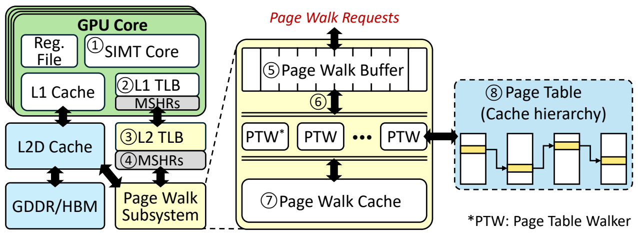


Figure 2: GPU architecture with virtual memory support

2 Background and Motivation

2.1 GPU Virtual Memory

Memory virtualization in GPUs is essential for supporting large address spaces and enabling unified memory access between CPUs and GPUs. GPUs employ a multi-level TLB hierarchy to reduce address translation overheads by caching virtual-to-physical address mappings. Typically, each SM has a private L1 TLB, while a large L2 TLB is shared among all SMs [45, 77].

Figure 2 depicts the address translation process in GPUs. When an SM issues global memory access, it first looks up the TLB to translate the virtual address to a physical address before accessing the cache hierarchy (①). On a TLB miss (②-③), the Miss Status Holding Registers (MSHRs) of each TLB hold the information of the missed request and forward it to the next TLB level or Page Walk Subsystem (④). Then, the Page Walk Subsystem buffers missed requests of L2 TLB into the Page Walk Buffer (PWB) (⑤). GPUs rely on a highly threaded Page Table Walker (PTW) to handle a large number of page table walk requests simultaneously generated by thousands of threads [75, 76]. In this paper, the terms 'page table walk' and 'page walk' are used interchangeably to refer to the process of translating a virtual address into a physical address by traversing the page table structure. The idle PTW selects a page walk request from the PWB and traverses page tables by issuing a memory request to the cache hierarchy (⑥). Since conventional GPUs adopt multi-level radix page tables [45, 60], the PTW sequentially accesses each level of page tables (⑧). Page Walk Cache (PWC) enables PTWs to skip accessing intermediate levels of page tables by caching recently accessed Page Table Entries (PTEs) (⑦).

2.2 Characteristics of GPU Address Translation

Modern GPUs leverage coalescing to maintain high throughput for regular workloads, merging a warp's memory accesses and translation requests when they fall within the same cache line or page [12, 81, 82]. However, this model easily fails for irregular workloads. The scattered memory accesses can cause a single warp instruction to trigger up to 32 distinct cache and TLB lookups, thereby decreasing the overall performance [6, 75, 76, 85, 86].

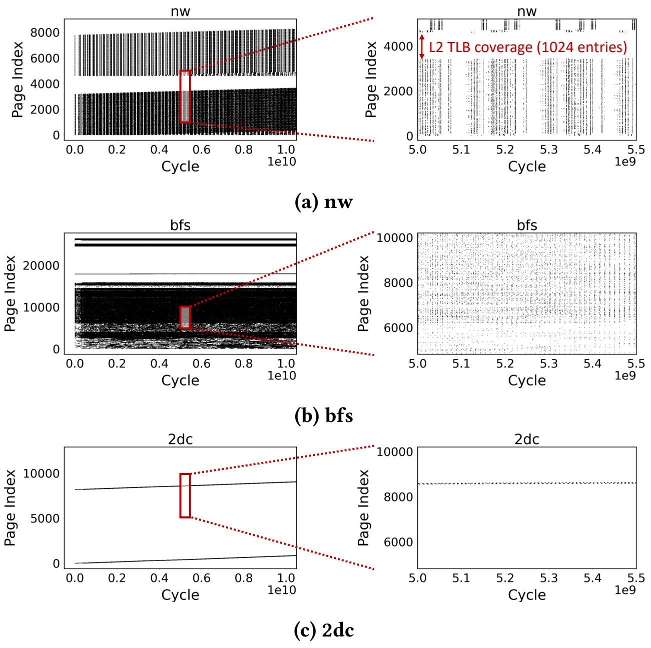


Figure 3: Access patterns of two irregular (nw [13], bfs [58]) applications and one regular (2dc [24]) application. For each application, the left figure shows the overall view of memory access pattern, while the right figure shows a zoomed-in view of the red-boxed region in the left. We use a 64KB page size in this experiment.

To analyze the memory access pattern in page granularity, we profile the addresses of global load/store instructions for three applications in the real GPU environment. Figure 3 shows the memory access pattern of nw, bfs, and 2dc in 64KB page granularity. The x-axis of each graph represents cycles, while the y-axis represents the page indexes. The 2dc shows a regular access pattern, sequentially accessing the contiguous region. Such a pattern allows GPUs to highly utilize their pipeline and memory bandwidth with a high TLB hit rate. In contrast, nw and bfs exhibit irregular memory access patterns, accessing a wide range of address space within a short time window. These access patterns easily exceed the L2 TLB coverage, resulting in frequent TLB misses and contention at PTWs.

To estimate page walk contention on a real GPU (NVIDIA A2000), we develop a microbenchmark that generates a variable number of concurrent page walks by issuing memory accesses from warps with one active thread, each accessing a distinct cache line. As shown in Figure 4, average memory access latency increases proportionally with the degree of concurrency. This trend directly

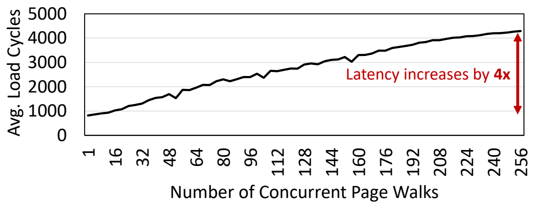


Figure 4: Average memory access latency of the NVIDIA A2000 as the number of concurrent page walks increases.

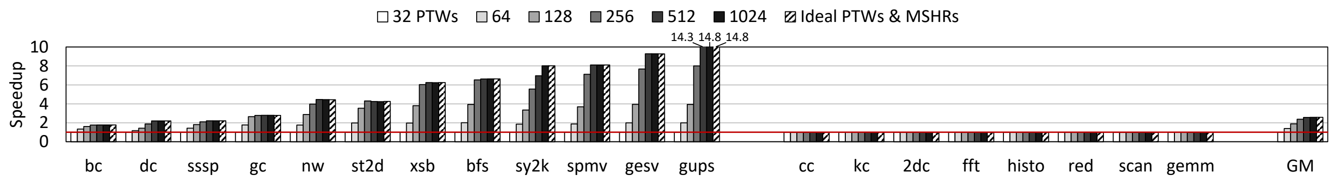


Figure 5: Impact of increasing PTWs on performance.

indicates contention; in an ideal system without bottlenecks, the latency would remain relatively constant regardless of the number of concurrent requests. At 256 concurrent walks, the latency increases by 4x compared to a single walk, confirming that page walk contention is a significant performance bottleneck in real GPUs.

Based on the observed memory access patterns and contention in page walks, we conduct a simulation-based study to evaluate the performance impact of PTW contention. Along with increasing the number of PTWs, we also enlarge the L2 TLB MSHR and PWB entries proportionally to accommodate the additional walkers. Detailed experimental configurations and benchmark information are in Table 3 and Table 4. Figure 5 shows the speedup of each application with increasing PTWs, normalized to the 32-PTW baseline. The results show that in the ideal case, where there is no page walk contention, the GPU achieves an average speedup of 2.58× over the baseline with 32 PTWs. Especially for irregular applications, which require more than 32 PTWs to reach peak performance, the GPU achieves an average speedup of 4.84× over the baseline. While regular applications are sufficient with 32 PTWs, irregular applications require PTWs from 256 to 1024 to fully resolve contention in page walks. These findings indicate that PTW contention significantly degrades overall performance by stalling address translation and leaving other resources, such as execution pipelines, underutilized.

While increasing the number of PTWs improves performance, it is not a scalable or practical solution due to excessive power and area overheads [23, 77, 86]. Increasing PTWs requires more entries in the L2 TLB MSHR and PWB, which are commonly implemented using fully associative structures, incurring significant power and area overheads [22, 27, 30, 50]. Therefore, a fundamentally different approach is required to increase the throughput of page walks.

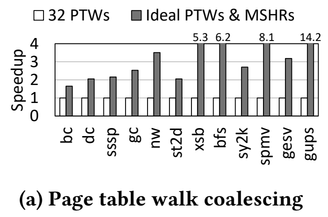
2.3 Mitigating Page Walk Overheads in GPUs

As discussed above, contention during page walks results in significant performance degradation, as it stalls the pipeline and underutilizes the cache hierarchy. Rather than simply increasing the number of PTWs, prior studies have explored alternative approaches from various perspectives to address this challenge.

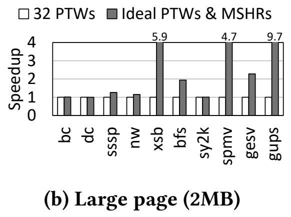
Coalescing Page Table Walk Requests. Neighborhood-Aware page table walk (NHA) [86] proposed a page table walk coalescing mechanism, which merges multiple page walk requests targeting the same cache line into a single request. Since GPU accesses cache hierarchy in a cacheline granularity (32-128B), this strategy can merge up to 16 page walk requests into a single one. However, NHA is only effective when currently pending page walk requests are fit into a single cacheline of the page table. When multiple page walk requests in the PWB cannot be merged (nw and bfs in Figure 3), there are substantial contentions in page walks.

Table 1: Comparison with prior works mitigating page walks

	Purpose	Approach	Flexibility	Need HW page walker?	Page walk throughput
NHA [86]	Reducing # page walks	Coalescing	No	Yes	~16×
Page Walk Scheduling [85]	Reducing warp divergence	Scheduling	No	Yes	X
HPT [32]	Removing pointer chasing	Hashed Page Table	No	Yes	X
SoftWalker (Ours)	Increasing page walk throughput	Software Thread	O (SW-based)	No	32×(# SMs)



(a) Page table walk coalescing



(b) Large page (2MB)

Figure 6: Impact of page table walk contention under large pages and page walk coalescing

Scheduling Page Table Walk Requests. Shin et al. [85] proposed a page walk scheduling mechanism to mitigate latency imbalance within a warp. They reduce the latency gap between the first and the last completed page walk requests in the same warp instruction by scheduling page table walk requests. Although this technique decreases the stall cycles of a warp, it cannot resolve the fundamental cause of page table walk contentions.

Hashed Page Table. Hashed Page Tables [29, 32, 48, 87, 88, 98] mitigate page walk overhead by replacing multiple page table steps with a hash-based single access. FS-HPT [32] adopted a Hashed Page Table (HPT) to the GPU virtual memory system to mitigate page walk overhead. By exploiting the insight that GPU HPT has a lower hash collision rate than CPU, it removed sequential accesses to each level of the page tables. However, FS-HPT aims to reduce memory accesses per page walk request, not increase the page walk throughput. Therefore, HPT-equipped GPUs will still suffer from contention in PTW.

TLB-Related Techniques. One of the strongest methods to reduce page walks is to increase the TLB hit rate. CoLT [74], Mosaic [6], and SnakeByte [49] extended TLB reach by coalescing PTEs of a contiguous virtual address region into a single entry. The coalesced entry can cover wider address regions than the system page size by introducing multiple valid bits in each entry or recursive merging. However, irregular workloads easily thrash TLB entries, reducing the effectiveness of coalescing. Meanwhile, Avatar [72] proposed a TLB speculation in GPUs. When an L1 TLB miss occurs, they speculatively calculate the physical address and validate the speculated address using the PTE embedded within a data cacheline. Even if

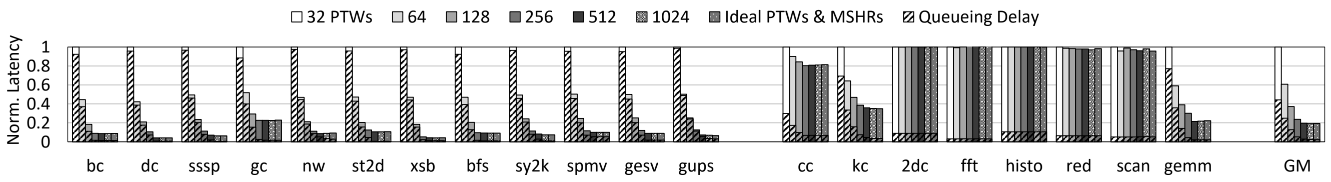


Figure 7: Breakdown of page table walk latency. The striped bar represents the queueing delay.

this approach eliminates L2 TLB accesses and page walks, it fundamentally requires a page walk when validating on an uncompressed cacheline, causing contention in page walks.

Common Constraints in Prior Techniques. As discussed above, prior techniques may still suffer from page walk contention for applications with large memory footprints and irregular access patterns at the page level. To verify that contention in page table walks remains even with prior techniques, we evaluate the speedup of page table walk coalescing [86] and large page (2MB) [6, 49, 74] as the number of PTWs increases. Since the input sizes of our benchmarks fit within the L2 TLB coverage for large pages, we select 10 benchmarks whose memory footprints can be scaled beyond the L2 TLB capacity and estimate the speedup by increasing their memory footprints. Figure 6 shows the impact of PTW contention when either page table walk coalescing (Figure 6a) or large pages is applied (Figure 6b). The results show that increasing PTWs still provides substantial performance improvements by mitigating page walk contentions, even in systems employing these techniques. These findings indicate that increasing the number of PTWs is beneficial and complementary to prior approaches.

Software-Based Approach. To enable scalable page walk throughput, we introduce *SoftWalker*, a software-based page table walk mechanism. Table 1 summarizes prior techniques aimed at mitigating page walk overheads in GPU virtual memory systems. Unlike prior approaches focusing on coalescing page walks or mitigating specific inefficiencies in the walks, *SoftWalker* directly targets the throughput limitation of hardware page walkers. *SoftWalker* achieves this by dynamically launching lightweight threads to perform page walks in response to L2 TLB misses. This approach offers scalability and flexibility, as the page walk process is implemented in software. Furthermore, *SoftWalker* eliminates the dependency on hardware page table walkers, unlike prior techniques.

3 A Software-Based Page Table Walk in GPUs

3.1 Key Concept

To mitigate the bottleneck resulting from a limited PTWs, we introduce *SoftWalker*, a software-based framework designed to handle page walks. The fundamental principle of *SoftWalker* is to dynamically launch multiple page walk threads on demand, thereby leveraging idle GPU cycles for page walk tasks, as illustrated in Figure 1. By providing sufficient page walk bandwidth on demand, *SoftWalker* effectively eliminates queuing delays caused by contention during page walks and resolves severe bottlenecks in page walks. In the following section, we will discuss key insights on applying software-based page table walks for GPUs.

3.2 The Dominance of Queueing Delay in Page Table Walk Latency

Contention among page walk requests increases page walk latency and stalls the GPU pipeline, degrading overall performance. In this paper, we define *page walk latency* as the sum of *queueing delay* and *page table access latency*. Queueing delay refers to the time between issuing an address translation request and the point at which a PTW begins processing it, while page table access latency denotes the time the PTW spends traversing the page table hierarchy. We observe that, for irregular applications, the dominant contributor to page walk latency is a queueing delay caused by contention in PTWs. Figure 7 illustrates how page walk latency changes with the number of available PTWs. The striped region of each bar represents the queueing delay. Our results show that, in the baseline configuration, queueing delay accounts for 95% of total page walk latency for irregular applications. This result indicates that contention at PTWs has a more significant impact on performance than the cost of individual page walks. As the number of cores continues to grow in modern GPUs, the contention in hardware PTWs is severely exacerbated, necessitating a scalable solution to alleviate the bottleneck and unlock performance for irregular applications.

Key Observation: For irregular GPU applications, the dominant factor contributing to page walk latency is queueing delays induced by contention in PTWs

3.3 Opportunity to Walk GPU Page Tables Using a Thread

Building on the key observation in Section 3.2, the question is how to scale page walk throughput without incurring extensive hardware cost. Previous CPU design explored software-managed TLBs, where TLB misses trigger exceptions handled by the Operating System (OS) [25, 31, 38, 57]. In such systems, a TLB miss causes the CPU to switch from user mode to kernel mode and invoke a TLB exception handler, which loads the missing PTE into the TLB before resuming execution of the faulting instruction. Finally, the OS returns control to the original context to replay the stalled instruction. However, frequent privilege transitions and heavyweight context switches make such approaches inappropriate for latency-sensitive CPU cores [19, 56].

In contrast, GPUs utilize fine-grained multithreading to interleave thousands of threads with minimal context switching overhead (as low as one cycle) [14, 55]. This massive thread-level parallelism inherent in GPU architectures presents a compelling opportunity to revisit software-based address translation. Specifically, GPUs

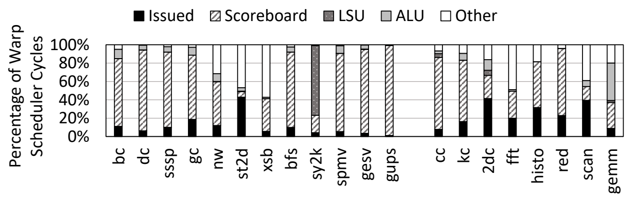


Figure 8: Breakdown of warp scheduler cycles of A2000

can potentially scale the throughput of page walks by parallelizing the walking process across threads. This unique capability opens the door to efficient software-managed page table walk mechanisms on GPUs, which can be designed to exploit thread-level parallelism to service multiple walk requests concurrently, thereby improving address translation throughput.

Key Insight ①: The massive thread-level parallelism and low context-switching overhead of GPUs make thread-based page table walks feasible and scalable.

GPUs rely on warp scheduling to efficiently hide memory latency by rapidly switching execution contexts among warps. Despite this capability, SMs often encounter significant stall cycles, predominantly due to long-latency memory operations and data dependencies. These stalls become severe in irregular workloads, where memory accesses exhibit minimal spatial locality and increased memory divergence, leading to ineffective warp-level coalescing and underutilized hardware resources [44, 86, 93, 97, 99].

To quantify the proportion of stall cycles during execution, we analyze warp scheduler cycles on a real GPU. Figure 8 presents the cycle breakdown for NVIDIA A2000 [61]. Except for issued cycles (shown in black), all other cycles indicate that the warp scheduler is unable to issue any instructions from the current active warps. For irregular applications, we observe that nearly 90% of the warp scheduler cycles are attributed to memory and scoreboard stalls, highlighting substantial underutilization of the GPU pipeline. These underutilized cycles represent a key opportunity: rather than remaining idle, they can be repurposed to perform page table walks in software. By utilizing these idle cycles for software-driven page table walks, GPUs can mitigate contention in page table walks and improve the utilization of their extensive computational resources.

Key Insight ②: Abundant stall cycles on irregular GPU applications provide enough room for executing a software-based page table walk.

GPU's massive parallelism, capability for rapid context switching, and enough idle core cycles provide a strong opportunity to realize a scalable software-based page table walk. The increased page walk bandwidth addresses the contention in handling PTWs, detailed in Section 3.2. Figure 9 provides a conceptual timeline view of three different page table walk scenarios.

In an ideal hardware scenario with sufficient PTWs (top), the total page walk latency consists solely of the page table access time alone. However, a realistic baseline with a limited number of PTWs (middle) suffers from heavy contention, causing queueing

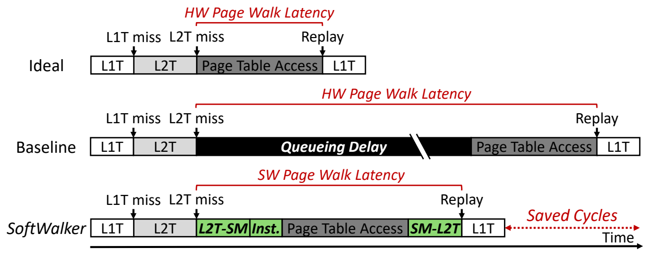


Figure 9: Page table walk latency comparison among ideal case with sufficient hardware PTWs (top), baseline with only a limited number of hardware PTWs (middle), and software-based approach (bottom). The green boxes represent additional latency for the software-based approach.

delay to become the dominant component (about 95%), thereby dramatically increasing the overall page walk latency. On the other hand, a software-based approach (bottom) effectively eliminates these contention-induced queueing delays. The key to improved page walk latency lies in a reasonable trade-off between slightly increased per-page walk latency and dramatically reduced total page walk latency. As shown in the figure, the software-based approach introduces small overheads for instruction execution and SM-L2 TLB communication (i.e., forwarding TLB misses to cores and returning page walk results to the TLB), making its intrinsic per-walk latency slightly longer than that of an ideal case. However, this slight increase in per-walk latency is negligible compared to the massive queueing delay it eliminates. As a result, by launching hundreds of walkers, *SoftWalker* mitigates the severe queueing delay that plagues the baseline system, yielding a substantial speedup.

Key Insight ③: The massive throughput of software-based page table walk eliminates the dominant queueing delay, compensating for the modest increase in individual walk latency.

In summary, the insights discussed above highlight the effectiveness of a software-managed GPU page table walker in improving page walk throughput. By leveraging thread-level parallelism and utilizing idle GPU cycles, this approach significantly reduces contention-induced queuing delays. In the following section, we detail the architecture and implementation of *SoftWalker*, and explain how it mitigates queueing delays in page table walks.

4 Implementation

4.1 Overview

SoftWalker is designed to efficiently handle GPU page table walks in software by leveraging GPU cores to parallelize and scale the translation process. Figure 10 illustrates a high-level overview of how page walk requests are managed and executed within *SoftWalker*.

The detailed operation flow of *SoftWalker* is as follows: Upon encountering an L2 TLB miss, the L2 TLB records the miss in its associated MSHRs. Then, the page walk request looks up PWCs and is forwarded to a Request Distributor (❶). The Request Distributor selects the core to process that request, and the SoftWalker Controller dedicated to each core fills the request into software-managed PWB

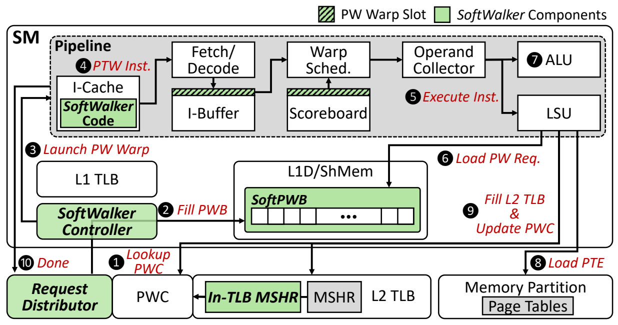


Figure 10: *SoftWalker* Architecture

(SoftPWB) (❷). Subsequently, the SoftWalker Controller initiates a specialized warp called a Page Walk Warp (PW Warp) to perform software-managed page table walks (❸). PW Warp fetches page walk code from the instruction cache and runs the instructions within the SM pipeline (❹-❺). The PW Warp's operations are as follows: loads page walk requests from PWB (❻), computes page table offsets (❼), gets PTE from in-memory page tables (❽), and fills L2 TLB and PWC (❾). Finally, once the PW Warp fills the L2 TLB entry, the SoftWalker Controller marks the operation as complete, freeing the PW Warp for the next page walk assignment (❿).

4.2 Page Walk Warp

SoftWalker performs software-based page walks by executing instruction sequences using GPU threads. To run these threads, the GPU must allocate threads on each SM. However, simply allocating dedicated threads for page walks reduces the number of threads available for the user kernel to be scheduled on an SM, potentially degrading application parallelism. Alternatively, sharing warp contexts—as explored in Assist Warp designs such as CABA [93]—could allow the reuse of existing warp resources. Nevertheless, this approach is unsuitable for address translation, where page walks involve accessing privileged virtual-to-physical mapping.

To resolve these challenges, *SoftWalker* introduces a *Page Walk Warp (PW Warp)*, a specialized warp that is structurally isolated from regular warps to each SM. Rather than consuming the limited number of user warp contexts, *SoftWalker* provisions dedicated architectural slots for the PW Warp, including an instruction buffer entry, scoreboard entries, and SIMT stack entries. The PW Warp is given the highest scheduling priority, as delays in page walks can stall the execution of other warp instructions and degrade overall performance. In addition to warp context resources, the PW Warp also requires a small number of registers. In our experiments, compiling the page walk code with a real GPU compiler revealed that a PW Warp requires only 16 registers, representing a negligible portion of the available register file. Recent studies have shown that GPU applications typically underutilize register files due to thread or thread block limits [1, 33, 34, 44, 45, 93, 99]. Leveraging this observation, *SoftWalker* allocates a small number of unused registers to the PW Warp with minimal impact on the resources.

The operation of PW Warp can be realized by the host driver to orchestrate PW Warps under a thread block model. Before the application kernel is dispatched, the driver preloads page walk instructions into device memory and launches a single, specialized block on each SM. This pre-launch reserves a minimal slice of core resources (e.g., registers, shared memory) so PW Warps can issue with zero admission delay and execute through the core's standard instruction pipeline like ordinary code. Once resident, the PW Warp enters a lightweight wait–execute loop governed by the SoftWalker Controller. As page walk requests are directed to an SM, the controller signals the warp scheduler to start scheduling a PW Warp. The loop persists for the lifetime of the application kernel, keeping translation capacity continuously available.

This design provides a secure and scalable mechanism to perform page table walks in parallel without limiting the occupancy of SM or compromising system isolation. We will discuss overheads or security issues associated with PW Warps in Section 5.

4.3 ISA Extension

To enable PW Warp to handle the entire page walk process, we extend GPU ISA to support 1) loading PTE using the physical address, bypassing TLB, 2) filling L2 TLB entry, 3) updating PWC entries, and 4) storing VPN that causes page fault to the Fault Buffer when it loads invalid PTE. We summarize the extended ISA in Table 2. First, `LDPT` is a memory load instruction that loads PTE to the destination register from the page table using the base address of each level of the page table. Second, `FL2T` is a special instruction that fills the L2 TLB entry with the last-level PTE that PW Warp loads from the page table. When L2 TLB receives the PTE that `FL2T` sent, it searches its MSHR and resolves the matching entry. Third, `FPWC` updates PWC entries right after it loads the PTE of each page table level. Since the L2 TLB sends page walk requests with the base address of the recently accessed level of the page table by referencing PWC, in most cases, each thread is only responsible for updating the Page Directory Entry (PDE). Finally, `FFB` is an instruction that stores the invalid PTE to the Fault Buffer for page fault handling [2, 42, 100]. These ISA extensions allow *SoftWalker* to complete the page walk process exploiting the GPU core pipeline, without relying on hardware page walkers.

Table 2: ISA Extension for *SoftWalker*

ISA	Description
`LDPT`	Load page table entry from the page table. This instruction bypasses accessing TLB
`FL2T`	Fill L2 TLB entry with the PTE.
`FPWC`	Fill Page Walk Cache entry.
`FFB`	Fill Fault Buffer with invalid PTE.

4.4 Page Walk Request Management

To manage page walk requests efficiently in *SoftWalker*, we need architectural support, especially for 1) distributing page walks from L2 TLB to each SM and 2) buffering page walk requests in each SM.

Request Distributor. First, we introduce a *Request Distributor* on the L2 TLB-side, which assigns each L2 TLB miss to a target core. The distributor maintains a per-core request counter to prevent request overflow during distribution and to ensure that page walk requests are assigned only to cores with idle PW Warps. The top half of Figure 11 represents the Request Distributor. Once the distributor selects a core based on each counter value (❶), it increments the counter associated with it (❷) and sends page table walk requests to

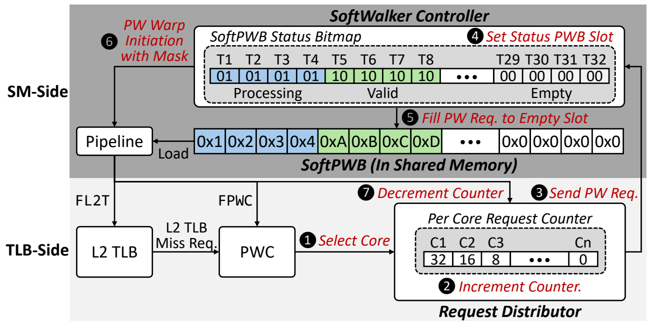


Figure 11: Management of page table walk requests

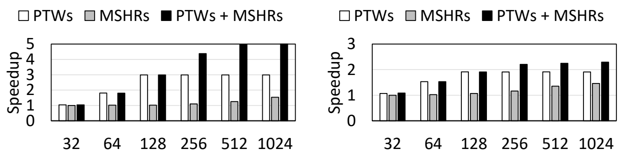


Figure 12: Speedup of irregular applications when scaling PTWs and L2 TLB MSHRs independently and jointly. "PTWs" scales the number of PTWs while fixing MSHRs, "MSHRs" scales MSHRs while fixing PTWs, and "PTWs+MSHRs" scales both. All results are normalized to a baseline with 32 PTWs and 128 L2 TLB MSHRs.

the selected core (❸). The counter is decremented when it receives the TLB fill request from that core (❼).

SoftPWB and SoftWalker Controller. Each SM must be able to accept and manage the page walk request that the Request Distributor sent. To support this, we introduce *SoftPWB*, a shared memory-based request buffer that holds pending page walk requests for that core's PW Warp. We repurpose a portion of the L1D/shared memory space as the SoftPWB so that the PW Warp can access the SoftPWB entries using standard load instructions. Each request consists of 1) a 33-bit VPN, 2) a 31-bit PFN of the page table base address derived from PWC, and 3) a 2-bit field for the current level of the page table, when assuming 49-bit virtual and 47-bit physical address [60]. Therefore, we reserve 96 bits for each SoftPWB entry.

To manage the buffer and orchestrate the execution of PW Warp, we introduce *SoftWalker Controller*. It tracks the status of each PW Warp thread using a compact bitmap called a SoftPWB Status Bitmap. Each entry in the bitmap contains 2 bits and encodes the state of a thread: invalid (no request assigned), valid (request ready for processing), or processing (currently executing a page walk). The bottom half of Figure 11 depicts the SoftPWB and SoftWalker Controller. Once the controller receives the page walk request, the controller updates the status of the 'invalid' entry to 'valid' by looking up the bitmap (❹) and fills the request to the corresponding SoftPWB position (❺). When the SoftWalker Controller detects a valid entry and the PW Warp is idle, it triggers a page table walk, updating the status of the corresponding entry from 'valid' to 'processing' (❻). Once the walk begins, the controller updates the status to "processing" and later resets it to invalid upon completion.

4.5 In-TLB MSHR

While *SoftWalker* increases page walk throughput by generating page walk threads, this enhanced throughput reveals another critical bottleneck: a limited number of L2 TLB MSHRs. If L2 TLB MSHR entries are fully occupied, the L2 TLB can not reserve requests from the L1 TLB (in this paper, we define it as MSHR failure). These MSHR failures prevent the system from fully exploiting thousands of page walkers [5, 21].

To see the impact of scaling PTWs and L2 TLB MSHRs independently and jointly, we estimate the performance of irregular applications: 1) fix L2 TLB MSHR entries as 128 and scaling PTWs, 2) fix PTWs as 32 and scaling L2 TLB MSHRs, and 3) scaling both. Each configuration is denoted as "PTWs", "MSHRs", and "PTWs + MSHRs" in Figure 12. The results, shown for both 64KB and 2MB page sizes in Figure 12a and 12b respectively, reveal that both resources are significant bottlenecks. With a 64KB page size, scaling only the PTWs achieves just 59.3% of the ideal performance, while scaling only the MSHRs is even less effective at 30.4%. Similarly, for a large 2MB page, with scaling only PTWs or only MSHRs reaching just 83.4% and 63.7% of the ideal case, respectively. This analysis suggests that it is crucial to scale both PTWs and L2 TLB MSHRs.

To expand the L2 TLB MSHR capacity at a low cost, we leverage underutilized L2 TLB entries for additional MSHR resources. We observe that only 2.4% of accesses result in L2 TLB hits across irregular applications, underscoring the severe underutilization of L2 TLB entries in such workloads. These findings motivate a new mechanism that better leverages the idle L2 TLB entries, enabling them to buffer outstanding TLB misses when the dedicated MSHR structures are already fully occupied [21, 96].

We propose *In-TLB MSHR*, a mechanism that extends MSHR capacity by allowing L2 TLB entries to act as temporary MSHR slots. Figure 13 shows the In-TLB MSHR design. Each entry equips a pending bit, enabling it to hold either a valid translation or metadata for a pending miss. The pending bit is associated with the valid bit to indicate three states (valid, invalid, MSHR) of each TLB entry. The operation of In-TLB MSHR is as follows: When a TLB miss occurs, and all MSHRs are occupied (❶), the TLB selects a victim entry based on its replacement policy (❷) and stores the metadata of the outstanding request (e.g., SM ID, Warp ID) in that entry (❸). We allow the in-TLB MSHR to reserve the same tag in a set index to support the MSHR merge [21]. Once the corresponding page walk completes (❹), the L2 TLB controller clears the pending bits of all tag-matching ways (❺). After that, it fills PTE information into one of the matching ways (❻) and resolves all corresponding misses (❼). In this way, *SoftWalker* can handle more concurrent misses even under high MSHR pressure.

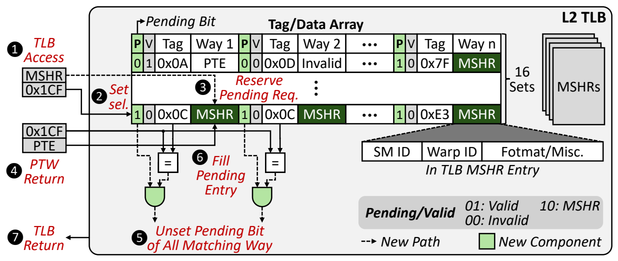


Figure 13: *In-TLB MSHR*

In-TLB MSHR boosts the effective capacity for in-flight translation requests without increasing the size of the dedicated MSHR structure. Because the design does not use TLB entries when existing MSHR entries are enough, it remains compatible with regular GPU workloads that rely on high L2 TLB hit rates.

4.6 Working Flow of *SoftWalker*

With this mechanism in place, a PW Warp is capable of handling L2 TLB misses by performing page table lookups in software, leveraging spatially and temporally underutilized resources on each core. First, the Request Distributor forwards L2 TLB misses to the SMs, and the SoftWalker Controller on each SM initiates a page walk. The SoftWalker Controller then marks the corresponding SoftPWB bitmap entry as 'processing' (changing it from 'valid') and launches the associated PW Warp to handle the request.

Figure 14 describes the abstraction of the instruction sequence that the PW Warp executes. The code sequence is preloaded onto the GPU memory by the GPU driver on the host side prior to kernel launch. Threads in PW Warp load page walk requests—such as virtual page numbers (VPNs), starting levels, and page table base addresses—from the SoftPWB into their register files, and decodes require information (*lines 3-6*). Then, they compute offsets for each level of the page table (*line 10*). Using the offset and page table base address, threads load PTE information by issuing the `LDPT` instruction (*line 13*). After loading PTE, if the PTE is invalid, the thread uses the `FFB` instruction to store it into the Fault Buffer (*lines 16-19*). If the PTE is valid, threads issue `FPWC`, which is a kind of store instruction to update PWC (*line 21*). Note that each thread starts its page walk from the PWC hit level as the Request Distributor consults the PWC before dispatching page walk requests. This process continues until the last-level page table is accessed (*lines 8-23*). When threads load the last level PTE, they complete the translation by inserting the final mapping into the TLB, issuing the `FL2T` instruction (*line 26*). The FL2T instruction also decreases the corresponding counter dedicated to the core at the Request Distributor. Finally, the SoftWalker Controller finalizes the PW Warp by updating the SoftPWB bitmap of matching threads from 'processing' to 'invalid'.

```
 1 int idx = threadIdx.x;
 2
 3 uint128_t pw_req  = softPWB[idx]; // load PW request
 4 uint64_t  pt_base = pw_req & 0x0FFFFFFF FFFFFFFF;
 5 uint64_t  vpn     = pw_req >> 64;
 6 int pt_level      = (pw_req >> 60) & 0xF;
 7
 8 while (pt_level < 5) {
 9   // Calculate page table offset for each level
10   int offset = (vpn >> ((pt_level-1)*9)) & 0x1FF;
11
12   // Access each level of the page table
13   pt_base = __load_page_table(pt_base, offset);      ▶ LDPT
14
15   // Page fault handling
16   if (__page_fault(pt_base)) {
17     __fill_fault_buffer(vpn, pt_level);              ▶ FFB
18     return;
19   }
20   // Update PWC
21   __fill_page_walk_cache(vpn, pt_level, pt_base);    ▶ FPWC
22   pt_level++;
23 }
24
25 // Fill L2 TLB entry
26 __fill_l2_tlb(vpn, pt_base);                         ▶ FL2T
```

Figure 14: Code Block for *SoftWalker*

5 Discussion

5.1 Security Implication

Resource Isolation of PW Warp. To ensure robust isolation, *SoftWalker* must safeguard its internal data structures from unauthorized access. Although GPU architectures provide intra-warp register communication via warp shuffle instructions (e.g., `__shfl_` [20, 66]), *SoftWalker* remains unaffected by potential attacks leveraging these instructions. This is because page walks are delegated to a dedicated PW warp, which operates in isolation from all other warps. Importantly, GPU hardware enforces strict register file partitioning, ensuring that register contents remain inaccessible across warps, CTAs, or kernel invocations [37, 59, 66, 68, 83]. Additionally, the translation metadata within SoftPWB is stored in shared memory, which is architecturally isolated from regular warps. An attacker thread block cannot access the data of a previously executed PW warp because shared memory is scoped to the thread block that allocated it; upon the block's termination, its entire memory mapping is invalidated. Moreover, an attacker block that runs concurrently on the same SM cannot invade the shared memory of a PW warp. This is because the shared memory of each thread block is mapped to a private and isolated logical address space [66]. Consequently, the GPU's hardware memory protection mechanism raises an exception on illegal memory access attempts, effectively preventing any out-of-bounds access.

Register File Uninitialization. The characteristic of *PW Warp*, which contains the page table's physical address in register files, induces a potential security vulnerability. Pustelnik et al. [79] recently demonstrated that several commercial GPU architectures (e.g., AGX, Adreno, RTX) omit register file initialization prior to shader execution. This enables leakage of stale values left in registers across kernel invocations, thus violating inter-kernel isolation. In the context of *SoftWalker*, uninitialized registers may expose sensitive metadata such as physical page table addresses to adversaries, which breaks down memory protection schemes. [28, 39]. *SoftWalker* can eliminate this vulnerability by adopting the guideline that inserts cleanup instructions after shader execution, which was proposed by Pustelnik et al. [79].

5.2 Hardware Overhead

SoftWalker requires additional storage in each SM associated with SoftWalker Controller and PW Warp. The controller requires 2 bits for each thread in PW Warp (total 64 bits) to track the status of page walk requests in the SoftPWB. To store PW Warp context, it requires a 64-bit instruction buffer, 126-bit scoreboard entry [44], and eight 160-bit (32-bit for 32-lane, 64-bit for PC, and 64-bit for RPC) SIMT Stack entry [44, 99]. Therefore, *SoftWalker* requires extra storage of 1470 bits (64+126+8x160) per SM.

In-TLB MSHR requires an additional 1-bit pending bit for each entry; a total of 1024 bits is required. In addition, to evaluate the overhead of In-TLB MSHR control logic, we implement it using Synopsys Design Compiler [89] with 28*nm* UMC standard cells [91]. The additional area overhead for In-TLB MSHR control logic is $0.0061mm^2$. Even if we synthesize it with a larger process (28*nm*),

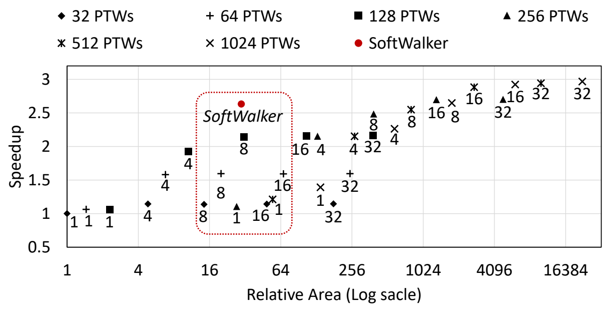


Figure 15: Speedup versus area overhead for various hardware configurations. Each data point is labeled with the number of PWB ports. Speedup and area overhead are normalized to the 32 PTWs with one PWB port.

the area overhead is extremely small compared to $628.4mm^2$, which is the GA102 full-chip area ($7nm$ process) [61].

5.3 Performance vs. Area

One potential solution for mitigating page walk contention is to increase the number of hardware page walkers. However, this approach is not cost-effective. Prior work [50] shows that 192 hardware page walkers with 18 PWB ports occupy 3.9% of the total GPU chip area. This overhead arises from the use of a Content-Addressable Memory (CAM) structure in the PWB, where scaling the number of entries and ports leads to super-linear area growth.

We evaluate the performance benefit of *SoftWalker* relative to its hardware cost by comparing its performance and area with those of hardware PTW scaling. Area overheads are estimated using CACTI [8, 80]. For the hardware-only approaches, we scale the number of PTWs and proportionally increase the PWB entries and L2 TLB MSHRs, which are implemented using CAM structure [50, 75, 76, 85, 86]. We also model the effect of varying the number of PWB ports for each PTW configuration. Figure 15 illustrates the trade-off between performance and relative area overhead for various configurations. The area efficiency of *SoftWalker* can be effectively evaluated by comparing it to hardware configurations with similar area overheads, highlighted by the red dashed box. Within this area budget (relative area ≈ 16–64), 32, 64, or even 128 PTWs achieve speedups of 1.1x to 2.1x. In contrast, *SoftWalker* achieves a higher speedup of over 2.6× within the same area budget. These results demonstrate that, for a given hardware cost, *SoftWalker* delivers substantially better performance than simply scaling hardware PTWs.

5.4 Hybrid Approach

While *SoftWalker* enables scalable software-managed page walks for irregular workloads, it may degrade the performance of regular applications with high TLB hit rates and low address translation pressure. This is because the regular applications have negligible queueing delay compared to irregular ones; the increased actual page walk latency could incur a slowdown. One of the promising solutions is retaining the existing hardware page walkers alongside *SoftWalker*. As a result, we propose a hybrid design, where Request Distributor prioritizes sending page walk requests to hardware page walkers as long as any are available; once no free hardware walkers remain, it redirects subsequent requests to software walkers. The hybrid version requires only adding a counter to track active page walk requests handled by hardware page walkers. With this design, regular workloads can utilize the hardware walkers as their primary translation mechanism, while PW Warps remain for irregular ones. We evaluate the implications of this hybrid design in Section 6.

5.5 Compatibility with Unified Virtual Memory

SoftWalker is compatible with Unified Virtual Memory (UVM) [84]. When a PW Warp encounters an invalid PTE (i.e., a page fault), it executes the dedicated `FFB` instruction to log the fault information into a fault buffer [2, 84, 100]. From the UVM driver's perspective, this behavior is equivalent to a page fault reported by a hardware page walker, enabling the existing page fault handling protocol to remain unchanged [9, 94].

Table 3: Experimental Setup

Component	Parameter
# of SMs	46 SMs
Clock Frequency	1500 MHz
Max. # of Warps	48 warps per SM
L1 TLB (per SM)	32 entries, 64KB page, 10 cycles, fully-associative, 32 MSHR entries, 192 merges per entry
L2 TLB (Shared)	1024 entries, 64KB page, 80 cycles, 16-way, 128 MSHR entries, 46 merges per entry
L1D Cache	128KB per SM, 40 cycles, 128B line (32B Sector)
L2D Cache	4MB, 180 cycles, 128B line (32B Sector)
Memory	GDDR6, 1750 MHz, 16 channels, total 448GB/s
Page Table	four-level radix page table
Page Walk Cache	32 entries, fully-associative
Page Table Walker	32 page table walkers
SoftWalker	32 page walk threads per SM, 32 *SoftPWB* entries per SM 128 L2 TLB MSHR entries, 46 merges per entry up to 1024 entry In-TLB MSHR

Table 4: Benchmarks

Type	Name	Abbr.	Footprint (MB)	L2 TLB MPKI	Required # PTWs
Irregular (Required # PTWs > 32)	betweenness centr [58]	`bc`	1194	9.0819	256
	degree centr [58]	`dc`	1138	26.17	512
	sssp [58]	`sssp`	1788	30.2808	512
	graph coloring [58]	`gc`	1294	13.7029	256
	nw [13]	`nw`	612	44.5329	512
	stencil2d [17]	`st2d`	612	4.8493	256
	xsbench [90]	`xsb`	360	57.9595	512
	bfs [58]	`bfs`	1396	22.1519	256
	syr2k [24]	`sy2k`	192	120.696	1024
	spmv [17]	`spmv`	288	2517.196	512
	gesummv [24]	`gesv`	226	1320.543	512
	gups [35]	`gups`	308	318.8202	1024
Regular (Required # PTWs ≤ 32)	connceted comp [58]	`cc`	2306	0.1309	32
	kcore [58]	`kc`	1152	0.5271	32
	2dconv [24]	`2dc`	1120	0.0767	32
	fft [17]	`fft`	610	0.077	32
	histogram [62]	`histo`	1124	0.0976	32
	reduction [62]	`red`	1124	0.3383	32
	scan [62]	`scan`	516	0.1458	32
	gemm [62]	`gemm`	288	0.0614	32

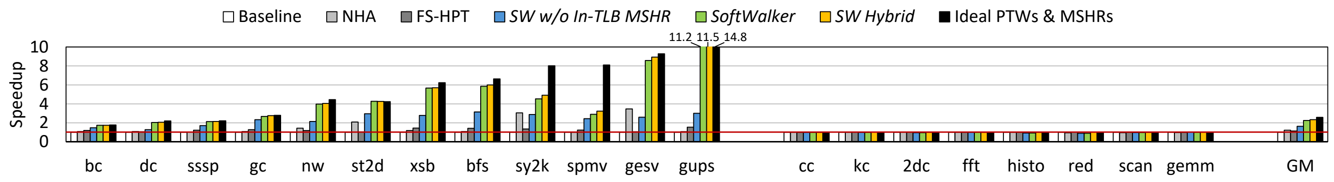


Figure 16: Overall Performance.

6 Evaluation

6.1 Methodology

Simulation Setup. We use Accel-sim v1.2.0 [41], a cycle-accurate GPU simulator configured similarly to an NVIDIA RTX 3070 GPU [61]. Table 3 summarizes our detailed simulation parameters. For the baseline architecture, we extended the simulator to support hierarchical TLBs, a complete page walk subsystem, and a multi-level radix page table. As described in Section 3.3, the total page walk latency is derived by adding page table access latency and the queueing delay. The hardware page table access latency in our simulation is not a fixed value but is dynamically measured by the memory system model, as page walks are a series of memory read requests whose latency depends on cache and DRAM behavior. We validate this approach by measuring the latency on an NVIDIA A2000 GPU, confirming our simulated latency (250-450 cycles) is consistent with real-world hardware performance (300-400 cycles) [72]. Our baseline system employs 128 L2 TLB MSHR entries and supports up to 32 concurrent hardware-managed page walkers [23, 53, 77, 78, 85, 86]. We adopt a 64KB page as the base page size, which is widely supported by conventional GPUs.

We implemented *SoftWalker* within the simulator by extending the ISA and incorporating the proposed architecture, including PW Warp, Request Distributor, SoftPWB, and In-TLB MSHR. The In-TLB MSHRs can accommodate up to 1024 outstanding misses when there is no free entry in the existing L2 TLB MSHR. Each SM equips one dedicated PW Warp and a 32-entry SoftPWB. The PW Warp runs assembly-level (i.e., SASS [65]) page walk code sequence, described in Figure 14. As defined in Section 3.3, the page walk latency of *SoftWalker* is the sum of page table access latency, additional communication latencies, and instruction execution latency. We model the communication latency as equal to the L2 TLB access latency, based on the assumption that the Request Distributor is located near the L2 TLB. The instruction execution latency is dynamically determined by the simulator's core pipeline model.

Benchmarks. We evaluate *SoftWalker* using a set of GPU workloads drawn from established suites [13, 17, 24, 35, 58, 62, 90]. As summarized in Table 4, we set the memory footprint of each benchmark to exceed the coverage capacity of the L2 TLB to ensure meaningful pressure on the address translation system. We classify workloads based on their page table walk concurrency requirement. Irregular workloads exhibit high L2 TLB MPKI and require a large number of concurrent page table walkers (more than 256) to hide queueing delays. In contrast, regular workloads with minimal TLB pressure operate efficiently with 32 PTWs.

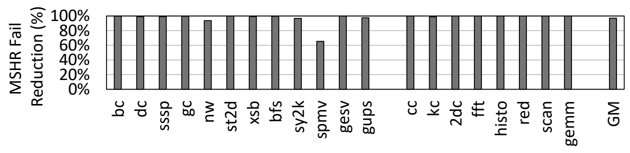


Figure 17: Percentage of L2 TLB MSHR failure reduction when enabling In-TLB MSHR relative to the baseline.

6.2 *SoftWalker* Result and Analysis

Speedup. Figure 16 shows the speedup of various GPU applications when applying Neighborhood-Aware address translation (**NHA**) [86], Fixed-Size Hashed Page Table (**FS-HPT**) [32], *SoftWalker* without In-TLB MSHR (**SW w/o In-TLB MSHR**), *SoftWalker* with In-TLB MSHR (**SoftWalker**), *SoftWalker* with hardware PTWs (**SW Hybrid**), and **ideal PTWs with ideal MSHRs** over our baseline GPU architecture. Among prior studies, we select NHA and FS-HPT as the state-of-the-art techniques since they aim to mitigate page table walk overhead. The NHA shows an average 1.22× speedup by eliminating page table walk requests that fit into the same cacheline (32B sector) of currently processed requests, while FS-HPT achieves an average 1.13× speedup over the baseline by eliminating the level dependency of traditional radix page tables. On the other hand, SW w/o In-TLB MSHR achieves an average 1.63× speedup over the baseline, as it can process up to thousands of page walks that NHA and FS-HPT can not.

However, increasing only PTWs shows limited performance improvement compared to the ideal case. This is because our baseline architecture equips a fixed number of L2 TLB MSHRs (128 entries), generating a fixed number of outstanding page walk requests. It limits the effectiveness of increased parallelism in page walks. With In-TLB MSHR, the *SoftWalker* can handle multiple concurrent page walks beyond the existing L2 TLB MSHR entries. Figure 17 shows the percentage of reduced L2 TLB MSHR failures when In-TLB MSHR is enabled compared to the baseline (32 PTWs). For irregular GPU applications, except for `spmv`, In-TLB MSHR almost eliminates the L2 TLB MSHR failures, enabling GPU threads to process thousands of page walks in parallel. Since `spmv` requires more MSHR entries than In-TLB MSHR can provide, it cannot completely eliminate the MSHR failures. Nevertheless, it reduces about 65% of MSHR failures and allows GPU threads to handle hundreds of page walk requests concurrently. Our experiment results show that In-TLB MSHR eliminates 95.3% of L2 TLB MSHR failures, enabling GPU threads to process thousands of page walks in parallel. As a result, *SoftWalker* with In-TLB MSHR achieves an average speedup of 2.24× (3.94× for irregular applications) over the baseline. In addition, compared to the prior works, *SoftWalker* shows an average speedup of 1.84× and 1.98× over NHA and FS-HPT, respectively.

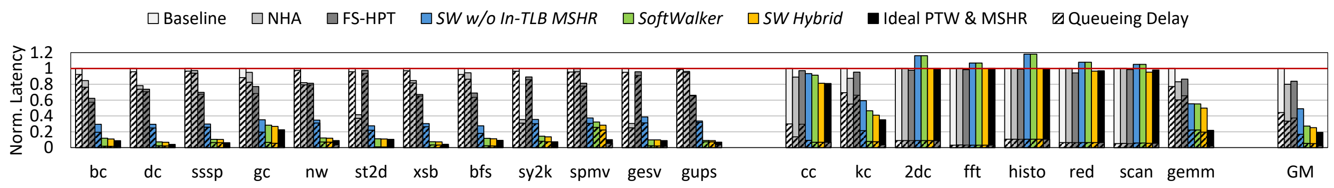


Figure 18: Page walk latency comparison. The striped bar represents the queueing delay.

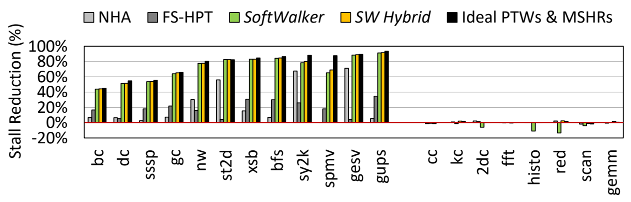


Figure 19: Stall reduction of warp scheduler cycles.

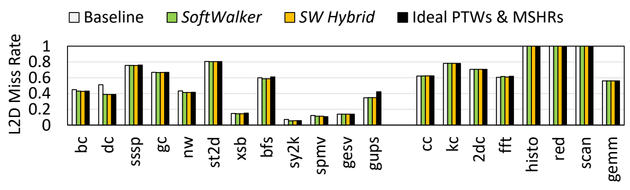


Figure 20: L2 data cache miss rate.

Page Table Walk Latency. Figure 18 shows the normalized page table walk latency of each configuration relative to the baseline. As mentioned in Section 3.3, we defined the page table walk latency as the sum of actual memory access latency and queueing delay. The NHA and FS-HPT still show high page table walk latency, reducing only 20% and 16% of page walk latency compared to the baseline, respectively. This is because those techniques cannot meet the required page walk throughput for irregular GPU applications. On the other hand, *SoftWalker* effectively eliminates almost all queueing delays of page table walks, reducing total page table walk latency by an average of 72.8% compared to the baseline. The reduction in the queueing delay is derived from 1) a reduction in L2 TLB MSHR contentions and 2) a reduction in PTW contentions.
Stall Reduction. *SoftWalker* exploits stall cycles to walk page tables using the PW Warp. By utilizing idle cycles, our proposed architecture can effectively eliminate contentions in page table walks, thereby reducing stall cycles caused by address translation. Figure 19 shows each application's percentage of stall reduction compared to the baseline when applying *SoftWalker*. For irregular applications, *SoftWalker* reduces an average of 71% stalls, resolving contentions in L2 TLB MSHRs and PTWs.
Impact on L2 Cache. The increased page walk traffic introduced by *SoftWalker* may raise concerns about potential L2 cache contention[2]. However, our analysis shows this is not the case; the L2 cache miss rate remains unchanged compared to the baseline, as shown in Figure 20. The primary reason is that the severe PTW contention in the baseline architecture leaves the memory system, including the L2 cache, heavily underutilized. Our experimental results show that irregular applications under the baseline configuration consume only 6.7% of total memory bandwidth on average. Similarly, we see no notable changes in the L2 cache MSHR failure ratio. Consequently, SoftWalker productively repurposes this idle memory bandwidth to handle page walks. Furthermore, since these irregular workloads already exhibit high L2 miss rates, the added page walk traffic is less likely to evict useful, long-lived data cachelines.

[2]The page walk traffic does not affect the L1D cache, as we assume PTEs are cached only in the L2 cache, following prior works [6, 7, 77, 78]

A Case for Regular Applications. The *SoftWalker* implementation incurs performance degradation in some regular GPU applications. Specifically, for `2dc`, `histo`, `red`, and `scan`, *SoftWalker* results in slowdowns of 4.3%, 8.6%, 10.9%, and 3.1%, respectively, compared to the baseline. This degradation originates from the additional page table walk latency introduced by the communication between the L2 TLB and each SM. As shown in Figures 18 and 19, regular applications without page table walk contention experience slightly increased page table walk latency (up to 18%) and stall cycles (up to 10%), leading to the observed slowdown.

Recognizing this potential performance issue for regular applications, as discussed in Section 5, we proposed an alternative design: the *SoftWalker* Hybrid version. This hybrid approach retains traditional hardware-structured PTWs alongside the numerous software-based walkers, specifically to benefit latency-sensitive regular applications. Our experimental results confirm that the *SoftWalker* Hybrid version effectively eliminates the performance gap observed with the pure SoftWalker, successfully reducing the page walk latency back to baseline levels.

6.3 Sensitivity Study

Comparison with Scaled Hardware Baselines. We compare *SoftWalker*'s performance against stronger hardware-centric baselines: 1) more hardware page walkers and 2) hardware page walkers with In-TLB MSHR. This analysis enables us to compare *SoftWalker* with a hardware configuration of comparable area cost (128 PTWs) and to identify the sources of its performance improvement.

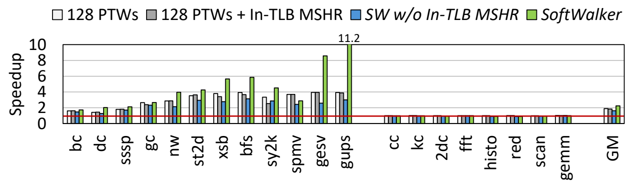


Figure 21: Speedup of *SoftWalker* and an iso-area hardware baseline (128 PTWs), evaluated with and without the In-TLB MSHR. All results are normalized to the baseline (32 PTWs).

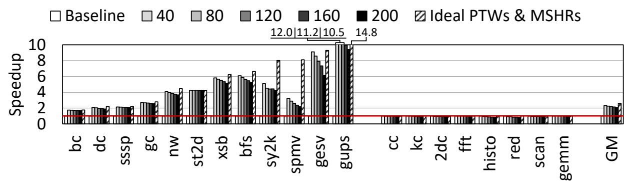


Figure 22: Impact of L2 TLB latency on *SoftWalker.*

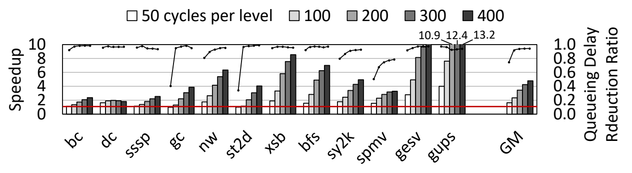


Figure 23: Impact of page table access latency on the speedup and queueing delay reduction ratio of *SoftWalker.*

First, focusing on an iso-area comparison, Figure 21 demonstrates that *SoftWalker* consistently outperforms the 128 PTWs by about 18.5% for irregular workloads. This demonstrates that, for a given hardware budget, the software-managed parallelism of *SoftWalker* is a more effective solution than simply scaling the number of hardware walkers. Second, the figure indicates that *SoftWalker*'s performance gain is not solely due to the In-TLB MSHR mechanism. Adding In-TLB MSHR to the baseline or to the 128-PTW configuration does not improve performance. Rather, it even causes a slowdown for `gc`, `xsb`, `bfs`, and `sy2k`, since there are fewer page walkers than the available MSHR entries. In such cases, In-TLB MSHR pollutes the L2 TLB by occupying a large portion of entries with long-lived pending translations. These results confirm that scaling both the number of PTWs and the MSHR capacity together achieves a significant performance gain, as analyzed earlier in Figure 12.

Varying L2 TLB Access Latency. Compared to the traditional hardware-based page table walk, *SoftWalker* requires additional communications between the L2 TLB and each SM. To see the impact of these additional latencies on *SoftWalker*, we estimate the speedup of *SoftWalker*, gradually increasing the L2 TLB access latency from 40 cycles to 200 cycles. Figure 22 shows the speedup of *SoftWalker* with various TLB latencies over the baseline. As the L2 TLB access latency gets longer, the performance of *SoftWalker* becomes lower. With the 40-cycle latency, *SoftWalker* shows 2.31× speedup, which is very close to the ideal (2.58×). Even though *SoftWalker* shows reduced speedup with a long, 200-cycle latency, it still shows 2.07×, which is comparable to the baseline. This is because the queueing delay of page table walk accounts for a large portion of total page table walk latency, even with the increased L2 TLB access latency, as we discussed in Section 3.3.

Varying Page Walk Latency. To evaluate *SoftWalker*'s sensitivity to memory access latency, we vary the per-level page table access latency from 50 to 400 cycles. Figure 23 shows the resulting speedup and queueing delay reduction for irregular applications. The speedup for each configuration is normalized to a corresponding baseline with 32 PTWs and the same per-level latency.

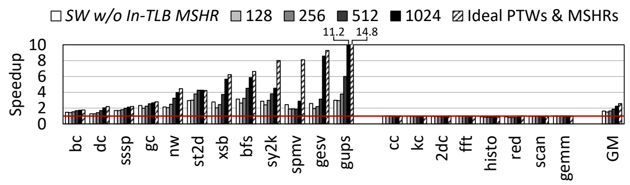


Figure 24: Impact of the number of In-TLB MSHR entries on the speedup of *SoftWalker* over the baseline.

The results show that *SoftWalker*'s speedup gradually increases as the per-level access latency increases. Specifically, when the per-level page table access latency is set to 50, 100, 200, 300, and 400 cycles, *SoftWalker* achieves the speedup of 1.6×, 2.3×, 3.5×, 4.2×, and 4.8×, respectively. A similar trend is observed in the reduction of queueing delay. This is because the lower page table access latency 1) incurs relatively low queueing delay and 2) reduces the performance sensitivity to page walks. Nevertheless, even with the lower page walk latency, *SoftWalker* achieves substantial speedup by eliminating a large portion of queueing delay.

Varying In-TLB MSHR Entries and Merges. In-TLB MSHR allows GPU threads to process more outstanding L2 TLB misses in parallel beyond the existing MSHR entries. As mentioned in Section 6.1, *SoftWalker* uses L2 TLB entries as storage for pending misses up to 1024 entries on demand. To evaluate the impact of In-TLB MSHR capacity, we measure the overall performance, varying the maximum number of In-TLB MSHR entries. (Note that the system allocates In-TLB MSHR entries only when all regular MSHR entries are full.)

Figure 24 illustrates the speedup of *SoftWalker* over the baseline, gradually increasing the maximum number of possible In-TLB MSHR entries. The results indicate that for maximum In-TLB MSHR entry counts of 0 (when In-TLB MSHR is not used), 128, 256, 512, and 1024, the respective average speedups over the baseline are 1.63×, 1.88×, 2.04×, 2.12×, and 2.24×, respectively. This improvement stems from In-TLB MSHR's ability to effectively resolve MSHR contention by repurposing underutilized L2 TLB entries to function as MSHRs. Consequently, most irregular applications demonstrate gradual performance gains as the In-TLB MSHR capacity expands.

Even with the existence of In-TLB MSHR, some applications do not reach the ideal performance as closely as other irregular ones. For `sy2k`, while In-TLB MSHR reduces almost all L2 TLB MSHR failures, the L2 TLB hit rate decreases as the maximum In-TLB MSHR increases. This leads to a sub-optimal performance for `sy2k`. Meanwhile, we found that `spmv`'s MSHR failure does not decrease further in over 128 entries because its access pattern frequently causes high contention within specific L2 TLB set indices. This per-set bottleneck limits the effectiveness of In-TLB MSHR for `spmv`. Despite these application-specific behaviors, In-TLB MSHR still offers a decisive improvement over the baseline by provisioning essential MSHR capacity on demand, thereby enabling substantial performance gains across a wide range of irregular workloads.

Large Page. Increasing the page size (e.g., 2MB) extends TLB coverage and reduces the number of page walk steps, effectively mitigating address translation and page walk overhead. However, some irregular applications still exhibit aggressive access patterns that extend beyond the increased page size. Figure 25 shows the speedup

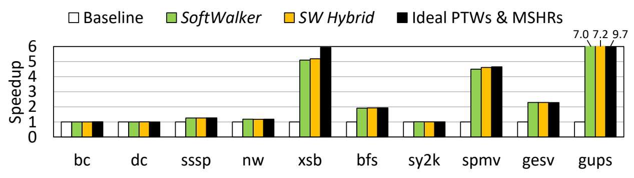


Figure 25: Speedup over baseline using 2MB pages.

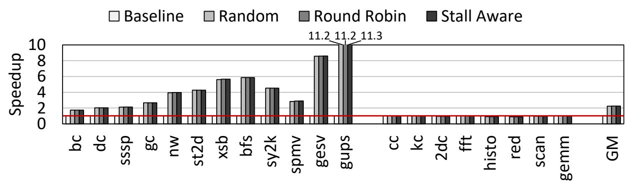


Figure 26: Speedup over baseline with alternative Request Distributor design.

of selected applications over the baseline when using a 2MB page size. We select 10 applications whose memory footprint can be expanded over the L2 TLB coverage (2GB in our configuration). Seven applications show performance improvement when applying *SoftWalker*. `sssp`, `nw`, `bfs`, `gesv` show 1.26×, 1.18×, and 2.29× speedup over the baseline, while `xsb`, `spmv`, and `gups` still show substantial speedup, 5.1×, 4.5×, 7.0×, respectively. These results suggest that *SoftWalker* can effectively reduce page table walk contentions, even with an increased page size.

Request Distributor Design. The Request Distributor assigns L2 TLB miss requests to SMs for processing. To evaluate the impact of its distribution policy, we compare three approaches: Random, Stall-Aware, and Round-Robin (our default choice) policies. Figure 26 compares the performance of *SoftWalker* under different Request Distributor policies, relative to the baseline. The results show no significant performance differences, as the high proportion of stall cycles in irregular applications ensures nearly all SMs have sufficient idle execution resources to handle page walks. Given that the specific policy is non-critical, we adopted the low-overhead round-robin approach for its effectiveness and simplicity.

7 Related Work

7.1 Leveraging Underutilized Resources

Prior research leveraged underutilized GPU resources to increase performance or power efficiency. CABA [93] utilizes assist warps to perform cache line compression during idle pipeline cycles, thereby alleviating memory bandwidth bottlenecks. Virtual Thread [99] introduces a novel CTA context-switching mechanism that increases Thread-Level Parallelism for memory-intensive workloads by exploiting unused register files and shared memory capacity. Other works, such as Reconfigurable Instruction Cache [45], Morpheus [18], and Linebacker [70], repurpose underutilized on-chip resources to expand cache or TLB capacity. Despite prior efforts, *SoftWalker* is the first to leverage idle GPU pipelines for resolving contentions in page table walks.

7.2 Warp Specialization

Warp specialization improves GPU efficiency by assigning warps to distinct functional roles. CUDA-DMA [10] offloads memory copy operations to dedicated warps to reduce interference with compute warps, while WASP [16] introduces hardware support for fine-grained warp role partitioning to exploit pipeline parallelism. These techniques typically target producer-consumer patterns within a thread block and rely on static assignment or compiler guidance. In contrast, *SoftWalker* introduces a dynamic and privileged form of specialization, launching dedicated warps (PW Warps) to handle address translation. This makes it uniquely suited for mitigating system-level bottlenecks, such as TLB miss-handling.

7.3 Mitigating Address Translation Overhead in Future GPU Architectures

While evolving GPU architectures meet increasing application demands, they also introduce new limitations in address translation. Trans-FW [54] and IDYLL [52] have sought to improve multi-GPU performance by mitigating address translation overheads arising from multi-GPU systems. MASK [7], DWS [77], and STAR [53] reduced contentions in page table walks and TLB caused by multi-tenancy. MGvm [78] and CLAP [71] attempted to mitigate remote access overhead caused by virtual memory systems in multi-chip GPU architectures [4]. *SoftWalker* complements these hardware-centric solutions, providing an additional avenue to reduce translation overheads through the efficient use of idle GPU resources.

8 Conclusion

We present *SoftWalker*, a scalable software-based page table walk mechanism that leverages GPU threads to address page walk contention for irregular GPU applications. Introducing lightweight architectural extensions—such as the PW Warp, SoftPWB, and In-TLB MSHR—*SoftWalker* enables massively parallel page walks and reduces contention-induced queueing delay. In addition, our hybrid design allows *SoftWalker* to operate with hardware walkers, providing low-latency page walk for regular applications. Experimental results show *SoftWalker* achieves a 72.8% reduction in page walk latency and a 2.24× speedup on average. *SoftWalker* demonstrates that software-managed page table walk is a scalable and efficient solution for future GPU memory systems.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was partly supported by The Ministry of Science and ICT (MSIT) under the Information Technology Research Center (ITRC) support program (No.II212052, 25%) supervised by the Institute of Information & Communications Technology Planning & Evaluation (IITP), the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government (MSIT) (No.10692981, 25% / No.00228970, 50%), and the BK21 FOUR Project. Sungbin Jang, Yongho Lee, Osang Kwon, and Seokin Hong are affiliated with the Department of Electrical and Computer Engineering. Donghyeon Kim is affiliated with the Department of Semiconductor Convergence Engineering. Juyoung Seok is affiliated with the Department of Semiconductor Display Engineering. Seokin Hong is the corresponding author.

References

[1] Mohammad Abdel-Majeed and Murali Annavaram. 2013. Warped register file: A power efficient register file for GPGPUs. In *2013 IEEE 19th International symposium on high performance computer architecture (HPCA)*. IEEE, 412–423.

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

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

[4] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 320–332.

[5] Mikhail Asiatici and Paolo Ienne. 2021. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 609–622.

[6] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 136–150.

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

[8] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

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

[10] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. 1–11.

[11] Gwangeun Byeon, Seongwook Kim, Hyungjin Kim, Sukhyun Han, Jinkwon Kim, Prashant Nair, Taewook Kang, and Seokin Hong. 2025. Avalanche: Optimizing Cache Utilization via Matrix Reordering for Sparse Matrix Multiplication Accelerator. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 1746–1759.

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

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

[14] Xi E Chen and Tor M Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 329–340.

[15] Compute Express Link Consortium 2023. *CXL 3.1 Specification*. Compute Express Link Consortium. https://computeexpresslink.org/cxl-specification/

[16] Neal C Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W Keckler. 2024. Wasp: Exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1–16.

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

[18] Sina Darabi, Mohammad Sadrosadati, Negar Akbarzadeh, Joël Lindegger, Mohammad Hosseini, Jisung Park, Juan Gómez-Luna, Onur Mutlu, and Hamid Sarbazi-Azad. 2022. Morpheus: Extending the last level cache capacity in GPU systems using idle GPU core resources. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 228–244.

[19] Francis M David, Jeffrey C Carlyle, and Roy H Campbell. 2007. Context switch overheads for Linux on ARM platforms. In *Proceedings of the 2007 workshop on Experimental computer science*. 3–es.

[20] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 73–84.

[21] Keith I Farkas and Norman P Jouppi. 1994. Complexity/performance tradeoffs with non-blocking loads. *ACM SIGARCH Computer Architecture News* 22, 2 (1994), 211–222.

[22] Yuan Feng, Yuke Li, Jiwon Lee, Won Woo Ro, and Hyeran Jeon. 2025. Heliostat: Harnessing Ray Tracing Accelerators for Page Table Walks. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 122–136.

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

[24] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 innovative parallel computing (InPar)*. IEEE, 1–10.

[25] Gary N Hammond, Koichi Yamada, Stephen G Burger, James O Hays, Jonathan K Ross, and William R Bryg. 1999. Software and hardware-managed translation lookaside buffer. US Patent 5,940,872.

[26] Sukhyun Han, Seongwook Kim, Gwangeun Byeon, Jihun Yoon, and Seokin Hong. 2025. Zebra: Leveraging Diagonal Attention Pattern for Vision Transformer Accelerator. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7.

[27] Mark D Hill and Alan Jay Smith. 2002. Evaluating associativity in CPU caches. *IEEE Trans. Comput.* 38, 12 (2002), 1612–1630.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[29] Won Hur, Jiwon Lee, Jaewon Kwon, Minjae Kim, and Won Woo Ro. 2025. HashScape: Leveraging Virtual Address Dynamics for Efficient Hashed Page Tables. *IEEE Trans. Comput.* (2025).

[30] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. 1999. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the 1999 international symposium on Low power electronics and design*. 273–275.

[31] Bruce Jacob and Trevor Mudge. 1997. Software-managed address translation. In *Proceedings Third International Symposium on High-Performance Computer Architecture*. IEEE, 156–167.

[32] Sungbin Jang, Junhyeok Park, Osang Kwon, Yongho Lee, and Seokin Hong. 2024. Rethinking page table structure for fast address translation in gpus: A fixed-size hashed page table. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*. 325–337.

[33] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*. 420–432.

[34] Naifeng Jing, Yao Shen, Yao Lu, Shrikanth Ganapathy, Zhigang Mao, Minyi Guo, Ramon Canal, and Xiaoyao Liang. 2013. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 344–355.

[35] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. 2015. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 international symposium on memory systems*. 223–234.

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

[37] Ni Kang, Ahmad Alawneh, Mengchi Zhang, and Timothy G Rogers. 2024. Concurrency-Aware Register Stacks for Efficient GPU Function Calls. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 686–699.

[38] Paul A Karger. 2007. Performance and security lessons learned from virtualizing the alpha processor. In *Proceedings of the 34th annual international symposium on Computer architecture*. 392–401.

[39] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*. 957–972.

[40] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.* 42, 4 (2023), 139–1.

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

[42] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-aware unified memory management in GPUs for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1357–1370.

[43] Junsu Kim, Jaebeom Jeon, Jaeyong Park, Sangun Choi, Minseong Gil, Seokin Hong, Gunjae Koo, Myung Kuk Yoon, and Yunho Oh. 2025. MOST: Memory Oversubscription-aware Scheduling for Tensor Migration on GPU Unified Storage. *IEEE Computer Architecture Letters* (2025).

[44] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. 2016. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 163–175.

[45] Jagadish B Kotra, Michael LeBeane, Mahmut T Kandemir, and Gabriel H Loh. 2021. Increasing gpu translation reach by leveraging under-utilized on-chip resources. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1169–1181.

[46] Osang Kwon, Yongho Lee, and Seokin Hong. 2022. Pinning page structure entries to last-level cache for fast address translation. *IEEE Access* 10 (2022), 114552–114565.

[47] Osang Kwon, Yongho Lee, and Seokin Hong. 2025. Improving Address Translation in Tagless DRAM Cache by Caching PTE Pages. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7.

[48] Osang Kwon, Yongho Lee, Junhyeok Park, Sungbin Jang, Byungchul Tak, and Seokin Hong. 2024. Distributed page table: Harnessing physical memory as an unbounded hashed page table. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 36–49.

[49] Jiwon Lee, Ju Min Lee, Yunho Oh, William J Song, and Won Woo Ro. 2023. Snakebyte: A tlb design with adaptive and recursive page merging in gpus. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1195–1207.

[50] Jiwon Lee, Myung Kuk Yoon, Ipoom Jeong, Yunho Oh, and Won Woo Ro. 2025. Marching Page Walks: Batching and Concurrent Page Table Walks for Enhancing GPU Throughput. In *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 626–639.

[51] Yongho Lee, Junbum Park, Osang Kwon, Sungbin Jang, and Seokin Hong. 2025. Buddy ECC: Making Cache Mostly Clean in CXL-Based Memory Systems for Enhanced Error Correction at Low Cost. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7.

[52] Bingyao Li, Yanan Guo, Yueqi Wang, Aamer Jaleel, Jun Yang, and Xulong Tang. 2023. Idyll: Enhancing page translation in multi-gpus via light weight pte invalidations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1163–1177.

[53] Bingyao Li, Yueqi Wang, Tianyu Wang, Lieven Eeckhout, Jun Yang, Aamer Jaleel, and Xulong Tang. 2024. STAR: Sub-Entry Sharing-Aware TLB for Multi-Instance GPU. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 309–323.

[54] Bingyao Li, Jieming Yin, Anup Holey, Youtao Zhang, Jun Yang, and Xulong Tang. 2023. Trans-fw: Short circuiting page table walk in multi-gpu systems via remote forwarding. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 456–470.

[55] Mat Loikkanen and Nader Bagherzadeh. 1996. A fine-grain multithreading superscalar architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*. IEEE, 163–168.

[56] Jeffrey C Mogul and Anita Borg. 1991. The effect of context switches on cache performance. *ACM SIGPLAN Notices* 26, 4 (1991), 75–84.

[57] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. 1993. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th annual international symposium on Computer architecture*. 27–38.

[58] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

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

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

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

[62] NVIDIA. 2024. NVIDIA CUDA Samples. (2024). https://github.com/NVIDIA/cuda-samples

[63] NVIDIA 2024. *NVIDIA GH200 Grace Hopper Superchip Architecture*. NVIDIA. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

[64] NVIDIA. 2024. NVIDIA High Performance Conjugate Gradient Benchmark (HPCG). https://github.com/NVIDIA/nvidia-hpcg

[65] NVIDIA. 2025. CUDA Binary Utilites. (2025). https://docs.nvidia.com/cuda/cuda-binary-utilities/contents.html

[66] NVIDIA. 2025. CUDA C++ Programming Guide. (2025). https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[67] NVIDIA. 2025. CUDA Toolkit Documentation. (2025). https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GRAPH.html

[68] NVIDIA. 2025. Nsight Compute Documentation. (2025). https://docs.nvidia.com/nsight-compute/index.html

[69] NVIDIA 2025. *NVML API Reference Guide*. NVIDIA. https://docs.nvidia.com/deploy/nvml-api/group__NvLink.html

[70] Yunho Oh, Gunjae Koo, Murali Annavaram, and Won Woo Ro. 2019. Linebacker: Preserving victim cache lines in idle register files of GPUs. In *Proceedings of the 46th International Symposium on Computer Architecture*. 183–196.

[71] Junhyeok Park, Sungbin Jang, Osang Kwon, Yongho Lee, and Seokin Hong. 2025. Leveraging Chiplet-Locality for Efficient Memory Mapping in Multi-Chip Module GPUs. In *Proceedings of the 58th Annual IEEE/ACM International Symposium on Microarchitecture*.

[72] Junhyeok Park, Osang Kwon, Yongho Lee, Seongwook Kim, Gwangeun Byeon, Jihun Yoon, Prashant J Nair, and Seokin Hong. 2024. A Case for Speculative Address Translation with Rapid Validation for GPUs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 278–292.

[73] Junbum Park, Yongho Lee, Sungbin Jang, Wonyoung Lee, and Seokin Hong. 2025. SPB: Towards Low-Latency CXL Memory via Speculative Protocol Bypassing. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7.

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

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

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

[77] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2021. Improving gpu multi-tenancy with page walk stealing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 626–639.

[78] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2022. Designing virtual memory system of mcm gpus. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 404–422.

[79] Frederik Dermot Pustelnik, Xhani Marvin Sass, and Jean-Pierre Seifert. 2024. Whispering Pixels: Exploiting Uninitialized Register Accesses in Modern GPUs. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 345–360.

[80] Divya Praneetha Ravipati, Victor M van Santen, Shivendra Singh Parihar, Yogesh Singh Chauhan, Preeti Ranjan Panda, and Hussam Amrouch. 2025. Cryo-CACTI: Cryogenic-Aware CACTI for Cache Modeling down to 10K in Advanced 7nm FinFETs. *IEEE Trans. Comput.* (2025).

[81] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 86–98.

[82] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 99–110.

[83] Mohammad Sadrosadati, Amirhossein Mirhosseini, Ali Hajiabadi, Seyed Borna Ehsani, Hajar Falahati, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. 2021. Highly concurrent latency-tolerant register files for GPUs. *ACM Transactions on Computer Systems (TOCS)* 37, 1-4 (2021), 1–36.

[84] Nikolay Sakharnykh. 2018. *Everything You Need to Know about Unified Memory*. https://www.nvidia.com/en-us/on-demand/session/gtcsiliconvalley2018-s8430/

[85] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling page table walks for irregular GPU applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 180–192.

[86] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-aware address translation for irregular GPU applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 352–363.

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

[88] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. 2023. Memory-efficient hashed page tables. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1221–1235.

[89] Synopsys 2025. *DC Ultra*. Synopsys. https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/dc-ultra-ds.pdf

[90] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[91] UMC 2022. *28 Nanometer*. UMC. https://www.umc.com/upload/media/05_Press_Center/3_Literatures/Process_Technology/28nm_Brochure.pdf

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

[93] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C Mowry, and Onur Mutlu. 2015. A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 41–53.

[94] Yueqi Wang, Bingyao Li, Aamer Jaleel, Jun Yang, and Xulong Tang. 2024. GRIT: Enhancing multi-GPU performance with fine-grained dynamic page placement. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1080–1094.

[95] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. 2022. A gpu-specialized inference parameter server for large-scale deep recommendation models. In *Proceedings of the 16th ACM Conference on Recommender Systems*. 408–419.

[96] Shaoxian Xu, Sitong Lu, Zhiyuan Shao, Xiaofei Liao, and Hai Jin. 2024. MiCache: An MSHR-inclusive Non-blocking Cache Design for FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 22–32.

[97] Jianchao Yang, Mei Wen, Dong Chen, Zhaoyun Chen, Zeyu Xue, Yuhang Li, Junzhong Shen, and Yang Shi. 2024. HyFiSS: A Hybrid Fidelity Stall-Aware Simulator for GPGPUs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 168–185.

[98] Idan Yaniv and Dan Tsafrir. 2016. Hash, don't cache (the page table). *ACM SIGMETRICS Performance Evaluation Review* 44, 1 (2016), 337–350.

[99] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. 2016. Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 609–621.

[100] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.