Towards Efficient and Scalable Multi-GPU Computing

by

Bingyao Li

University of Pittsburgh

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Bingyao Li

It was defended on

March 10, 2024

and approved by

Dr. Xulong Tang, Department of Computer Science

Dr. Bruce Childers, Department of Computer Science

Dr. Youtao Zhang, Department of Computer Science

Dr. Jun Yang, Department of Electrical and Computer Engineering

Towards Efficient and Scalable Multi-GPU Computing

Bingyao Li, PhD

University of Pittsburgh,

In the past decade, Graphics Processing Units (GPUs) have rapidly evolved as one of the most popular computing platforms to provide significant acceleration in machine learning, graph processing, scientific computing, and VR/AR. The ever-growing application complexity and input dataset sizes have driven the popularity of multi-GPU systems as desirable computing platforms. While employing multiple GPUs intuitively exposes substantial parallelism for the application acceleration, the delivered performance rarely scales with the number of GPUs. Specifically, the execution efficiency suffers expensive address translations in multi-GPUs. The address translation process involves hierarchical TLB lookups and expensive page table walks. More importantly, the translation process involves non-local components such as TLB and page table walk on the CPU side in an I/O memory management unit (IOMMU) organized multi-GPU system, generating non-uniform and unpredictable long latencies. Such remote resources can be severe bottlenecks than local resources since i) accessing remote components experiences long latencies and ii) the aggregated requests from multiple GPUs cause severe contention. Furthermore, the data-sharing nature of GPU applications requires page migration between GPUs to mitigate non-uniform memory access overhead. Unfortunately, frequent page migration incurs substantial page table invalidation overheads to ensure translation coherence.

My research analyzes and optimizes the address translation on multi-GPU systems toward scalable executions. First, my research redesigns the TLB hierarchy and proposes i) "least-inclusive" TLB hierarchy and ii) hardware-supported address translation sharing with peer GPUs. Second, my work focuses on uncovering the bottlenecks and exploring opportunities in page table walking (PTW) in multi-GPUs. Finally, I further investigate the effects of frequent page migration invalidations in multi-GPU systems and propose a software-hardware co-design to mitigate the page table invalidation overhead and improve overall application performance.

Table of Contents

List of Tables

List of Figures

Preface

1.0 Introduction

1.1 Problem Statement

In the past decade, Graphics Processing Units (GPUs) have rapidly evolved as one of the most popular computing platforms to provide significant acceleration in machine learning [64, 80], graph processing [66, 29, 65], scientific computing [46, 74], and VR/AR [26, 84]. Such popularity has motivated a comprehensive literature of GPU optimizations focusing on "single-application-single-GPU" and "multi-application-single-GPU" execution paradigms [14, 22, 67, 68, 11]. Recently, the ever-growing application compute-intensity and memory-intensity have driven a shift of attention from single GPU systems to multi-GPU systems as the capability of a single GPU is no longer able to catch up with the application requirements. As a result, multiple GPUs are employed to collaboratively execute the applications, bringing "single-application-multi-GPU" execution paradigm. Moreover, multi-tenancy (i.e., applications) has become a general feature on server-class GPUs to accommodate concurrent execution of applications with a variety of characteristics, bringing the "multi-application-multi-GPU" execution paradigm.

To cater to the application execution characteristics, GPU vendors have explored different incarnations of multi-GPU systems, including NVIDIA's DGX [49] and Intel Xe [32]. Modern multi-GPU systems generally employ unified virtual memory (UVM) to simplify programming and improve application portability and compatibility [3, 4, 48, 52]. UVM allows GPUs to access data residing in remote physical memory through universal pointers. However, the potential and the delivered application performances of UVM-enabled multi-GPUs are constrained by i) expensive address translation process and ii) non-uniform memory access (NUMA) overheads arising from data sharing and communication across GPUs. While substantial prior works have investigated the NUMA overheads [8, 45, 47, 88], the address translation receives little attention in multi-GPUs. Specifically, the translation that misses the GPU local TLB hierarchy must go through local multi-level page table walk handled by GPU MMU (GMMU). If the page is invalid in the local page table, a far fault is

Techniques	Reduce invalidation	Improve PTW	Write intensive	Support Multi-App	Support Multi-GPU
TLB optimizations [87, 37] [78, 61, 57, 59, 75, 12]	No	Yes	No	No	No
PW-cache design [18, 58, 43, 44]	Partial	Yes	Yes	Partial	No
Page walk scheduling [62, 67]	No	Yes	Yes	Partial	No
Large page [9, 60, 56]	Partial	Partial	No	No	No
Dynamic page migration [11]	No	No	Yes	No	Yes
Page replication [24, 47]	Yes	No	No	No	No
Our approach	Yes	Yes	Yes	Yes	Yes

Table 1: Comparison with prior techniques.

generated and it experiences a long latency in accessing the TLB and page table located in the CPU I/O memory management unit (IOMMU). Such latency gets longer when multiple GPUs simultaneously compete the shared IOMMU (which is common in "single-application-multi-GPU" and "multi-application-multi-GPU" executions), leading to performance degradation. For example, the translation process can occupy up to 50% of the total execution time [37, 17, 13].

Many prior works have explored address translation optimizations in CPUs and GPUs. Table 1 summarizes these optimizations in the literature. First, most TLB optimizations (e.g., range-based TLB [87], clustering TLB [59], compression [75], and TLB probing [12]) focus on single-GPU/CPU executions and cannot be effectively applied to the multi-GPU environment. While these techniques effectively improve TLB reach, they cannot mitigate the frequent local page faults in multi-GPUs and help little with the page sharing caused page migration. Moreover, these TLB optimizations rely on continuous, stride, and predictable memory access patterns. While such regular patterns might be observable in each individual application, it is rarely observed at the shared IOMMU TLB when multiple applications run concurrently, mainly due to the interference among applications. Second, existing page walk cache optimizations [18, 58, 43, 44] can accelerate the page table walks for invalidation requests by reducing page walk cache misses. However, the substantial invalidations thrash the page walk cache and lead to frequent eviction of useful entries required by existing demand TLB miss requests. Third, page walk scheduling works [62, 67] enable a trade-off between page table walk throughput and fairness. However, these works are not applicable to page migration invalidations as they do not help with the significant amount of invalidations

generated by intensive page sharing across GPUs. Fourth, a large page [9, 60, 56] increases the TLB reach and reduces the contention in page table walk in a single GPU. However, in multi-GPU, when a large page is frequently shared among different GPUs, the false sharing may introduce more page migrations and additional invalidation requests. Fifth, the dynamic page migration [11] is effective in reducing remote data access, however, the PTE invalidation overheads caused by page migration have yet to be addressed. Finally, page replication [24, 47] enables pages to be accessed locally without page migration. However, it is not feasible for applications where the shared pages have read-write properties, as write operations still require invalidating PTEs and pages. Further, with substantial page sharing among multiple GPUs, page replication is not scalable [39, 63].

These limitations of the prior works motivate us to rethink the address translation designs in the context of multi-GPU environment:

> *Can we holistically orchestrate all steps of address translation to achieve better performance?*

In this thesis, we systematically investigate and optimize the address translation in multi-GPU systems. We observe and quantify latency overheads in the translation process. To mitigate these overheads, we propose Least-TLB to reduce the translation redundancy and improve the TLB reach in multi-GPUs, Trans-FW to improve the page table walk performance in multi-GPU executions, and IDYLL to mitigate the page table invalidation overhead in multi-GPU. In our future work, we will investigate the address translation of multi-application execution in multi-instance GPUs to accommodate concurrent execution of applications with a variety of characteristics, bringing the "multi-application-multi-GPU" execution paradigm.

1.2 Contribution Overview

- We first target ATS-organized multi-GPU systems, in which the TLB hierarchy consists of local (i.e., within GPU) and remote (i.e., in IOMMU) counterparts. We first conduct a comprehensive investigation of the GPU local TLB and IOMMU TLB performances

and their impact on the overall application performance. We observe that the conventional "mostly-inclusive" TLB designs [15, 86] involve multiple deficiencies when used in the "local-remote" multi-GPU TLB hierarchy. Then, we propose *least-TLB*, which comprises several inter-related optimizations to improve the IOMMU TLB performance. Specifically, we propose i) "least-inclusive" TLB design to reduce the translation redundancy and improve the TLB reach; ii) hardware-supported address translation sharing with peer GPUs in their local TLB; and iii) IOMMU TLB spilling to reduce the contention when multiple applications execute concurrently.

- We then investigate the page table walk performance in MMU-based multi-GPUs. We identify three latency penalties in the multi-GPU address translation process. We propose Trans-FW to improve the page table walk performance in multi-GPU executions. First, short circuiting is employed in GMMU by early sending local page faults to host MMU. Second, we leverage remote GPU to supply translation requests instead of waiting in the host when doing so is beneficial.
- We quantitatively show that contention between demand TLB miss requests and page migration-induced page table invalidation requests significantly limits multi-GPU performance. We propose IDYLL which employs two key mechanisms to minimize this contention. First, we reduce the number of unnecessary invalidation requests by employing an "in-PTE" directory that leverages unused bits in the page table entry to record which GPUs hold valid address mappings for the corresponding PTE. Second, we minimize interference between invalidation requests by employing lazy invalidation which exploits spatial locality in page table updates by batching multiple invalidation requests with nearby virtual addresses. We design a hardware structure called Invalidation Request Merging Buffer (IRMB) to temporarily hold these batched invalidation requests and lazily write them back to the page table. Specifically, the IRMB is checked in parallel with the GPU L2 TLB lookup. For those TLB misses, if they are found in the IRMB, the corresponding page table walks are removed to avoid accessing stale PTEs and to ensure the correctness of the translation.

2.0 Background

2.1 Multi-GPU Architecture

We focus on UVM-managed discrete multi-GPU systems where the GPUs have unified virtual memory address space and use pointers to access memory in remote GPUs. The GPUs are connected using a high-bandwidth interconnect such as PCIe or NVLink. Figure 2 shows the architecture details of the targeted baseline multi-GPU system. Each GPU consists of multiple Shader Arrays (SAs), each of which further contains multiple compute Units (CUs), a.k.a., SMs in NVIDIA terminology. Every CU has its own private L1 data cache (L1V$); all CUs within an SA share the L1 scalar cache (L1S$) and the L1 instruction cache (L1I$). There is also a larger unified L2 cache that is shared among all the SAs. Each GPU also features a multi-level TLB hierarchy to accelerate address translation: each CU has a private fully associative L1 TLB, and all CUs (in all SAs) share the L2 TLB.

There are two widely used multi-GPU architectures, ATS-managed multi-GPU and MMU-managed multi-GPU. In ATS-manages multi-GPU, an Input/Output Memory Management Unit (IOMMU) on the CPU side is shared by all GPUs and has its own TLB. It is used to handle the requests generated from all the GPUs. Each GPU has its own local memory. However, the page tables are centralized in the CPU memory and controlled by the CPU [16, 55, 82]. Therefore, address translations that miss in the GPU L2 TLBs will be forwarded to the IOMMU TLB for "remote" IOMMU TLB lookup or page table walk (PTW). In MMU-managed multi-GPU, each GPU has its own local memory as well as a local page table. The GPU Memory Management Unit (GMMU) handles GPU page table walks. A GMMU typically consists of (i) a page walk queue for buffering the translation requests, (ii) a page walk cache holding the entries of page table levels used in recent translation requests, and (iii) multi-threaded page table walker that handles multiple translation requests concurrently. In MMU-managed multi-GPU systems, the UVM driver on the CPU side is responsible for handling all GPU far faults: the UVM driver maintains a centralized page table holding the up-to-date address translations for all GPUs.

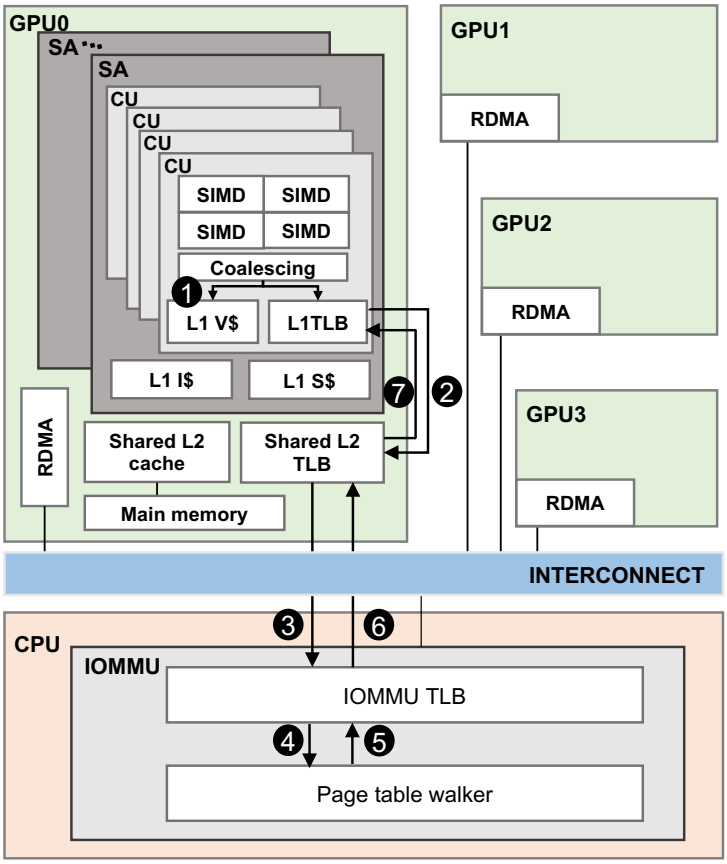

Figure 1: Baseline GPU architecture.

2.2 Address Translation in GPUs

Figure 1 illustrates the ATS-managed address translation process. Memory access requests from the same wavefront are first coalesced by the memory coalescing unit. Then, the coalesced accesses are sent to L1 scalar cache and L1 TLB for parallel lookup (❶), assuming a virtually indexed physically tagged cache-TLB design. If the lookup misses in the L1 TLB, the translation request is forwarded to the shared L2 TLB (❷). If again the request misses in the L2 TLB, the GPU will generate an Address Translation Service (ATS) packet and send it to the IOMMU (❸). Note that, each GPU in our modeled baseline has its own local device memory but the page tables are shared on the CPU side [11, 72]. Upon receiving the ATS packet, the IOMMU will check its TLB to see if the translation is present. If it is

present, the translation will be returned to the GPU. If not, the IOMMU will trigger the multi-threaded PTWs to traverse the entire page table (❹). Once the translation is found in the page table, it is then populated into the IOMMU TLB (❺) and sent to the requesting GPU. Upon receiving, the GPU will populate the translation to its L2 TLB (❻) as well as L1 TLB (❼). Note that, when the IOMMU PTWs detect page faults, the IOMMU will send an ATS response to notify the GPU about this failure. Then, the GPU will send a request called Page Request Interface (PRI) to the IOMMU. The IOMMU records PRI requests from different GPUs in a queue and interrupts the CPU for page fault handling. As the page fault handling incurs significant latencies, the IOMMU typically uses batching to amortize the overhead among multiple PRI requests [7, 16].

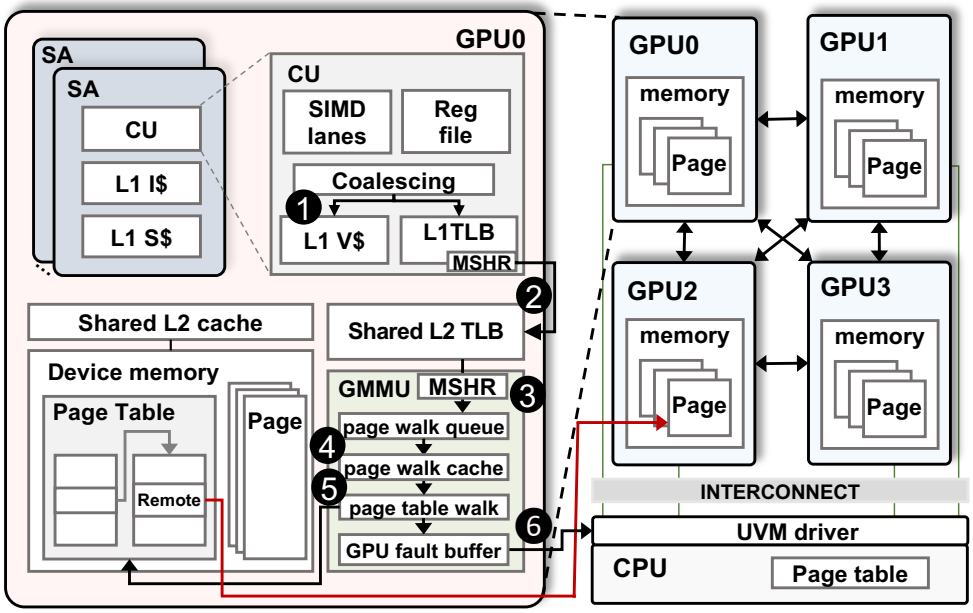

Figure 2: Baseline GPU architecture.

Figure 2 illustrates the MMU-managed address translation process. Upon a memory request, the L1 cache and the L1 TLB lookups are performed in parallel, assuming the L1 cache is virtually indexed and physically tagged (❶). If the request misses in the L1 TLB, it first checks the L1 Miss Status Holding Register (MSHR), and the request is sent to the L2 TLB for lookup upon MSHR miss (❷). Similarly, requests missing in the L2 TLB are further sent to the GMMU (❸) to perform page table walks. The request is temporarily stored in

the page walk queue if there are no available page walk threads. When performing a page walk, GMMU generates several memory requests to access the local page table (❺). To take advantage of the temporal locality of the page table, the GMMU maintains page walk cache for each level of the page table (❹). If the page walk fails, a far fault is propagated to the GMMU and held in the GPU Fault Buffer [4, 3]. Then, the GMMU notifies the host UVM driver about the far fault by generating an interrupt (❻). The UVM driver, on the host side, fetches the fault information, groups faults into batches, and caches it on the host (the batch size is 256 [50]), and later resolves the fault using the centralized page table. After these steps, it initiates the target data transfer and updates the requesting GPU's local page table. Eventually, the GPU re-performs the address translation after the far fault is resolved.

2.3 Page Migration Scheme

Frequent accesses to remote data can incur significant performance bottlenecks since remote GPU memory access bandwidth over the interconnection network can be an order of magnitude lower than local GPU memory access bandwidth. One approach to address the remote data access performance bottleneck is to migrate pages from remote memory to local memory. There exist multiple page migration schemes:

On-touch migration: Every time a GPU accesses a page residing in another GPU, the page is migrated into the requesting GPU memory. While it guarantees page accesses are going to local pages, frequent "ping-pong" migration may decrease the performance.

First-touch migration: A page is migrated from the CPU to the GPU on the GPU's first access. After this initial migration, this page is pinned on that GPU, i.e., it will never be migrated to another GPU. Compared to on-touch migration, first-touch migration does not incur frequent page migrations and thus avoids the invalidation/migration overheads. However, remote memory accesses may incur high latency.

Access counter-based migration: Recent Nvidia GPUs (Volta and newer generations [48]) use page access counters to delay the migration of pages. On each remote memory access, the corresponding page access counter is incremented. A page migration is only performed

when this counter reaches a certain threshold (e.g., 256 [50]). There are four steps for access counter-based page migration. First, the GPU initiates a migration request to the UVM driver. Second, the UVM driver broadcasts the invalidation requests to every GPU in the system since the UVM driver is unaware of which GPU(s) have the translation of this page. Third, upon receiving invalidation requests, each GPU performs TLBs shootdown and invalidates the page table entries to ensure translation coherence [11]. The PTE invalidation is performed in a way similar to the conventional address translation procedure: the GMMU starts a page walk which may contend with other page walk requests. Finally, the UVM driver initiates the data transfers. This scheme incurs fewer remote memory accesses compared to first-touch migration, and fewer page migrations compared to on-touch migration.

3.0 Improving Address Translation in Multi-GPUs via Sharing and Spilling aware TLB Design

3.1 Motivation

3.1.1 Baseline Configuration and Workloads

We use MGPUSim [72] to conduct our characterization and later evaluate our proposed least-TLB design. MGPUSim models AMD multi-GPU system and is validated against AMD R9 Nano GPUs [6]. We heavily extended MGPUSim by adding the IOMMU module with a shared TLB to handle ATS and PPR requests.

Table 2: GPU system Configuration.

Module	**Configuration**
CU	1.0 GHz, 64 per GPU
L1 Vector Cache	16 KB, 4-way
L1 Inst Cache	32 KB, 4-way
L1 Scalar Cache	16 KB, 4-way
L2 Cache	256 KB, 16-way
DRAM	512 MB
L1 TLB	16 entries, 16-way, 1-cycle lookup latency, CU private, LRU replacement policy
L2 TLB	512 entries, 16-way, 10-cycle lookup latency, CUs shared, LRU replacement policy
IOMMU TLB	4096 entries, 64-way, 200-cycle lookup latency, GPUs shared, LRU replacement policy
Page table walk	8 shared page table walker, 500-cycle latency [75]

3.1.1.1 Baseline GPU configuration

In this work, we target an ATS-managed 4-GPU system with a shared IOMMU. It is important to emphasize that our approach is also applicable to multi-GPU systems with more GPUs. Table 5 shows the baseline GPU configurations. The page size is set to 4KB. The baseline TLB hierarchy includes per-CU private L1 TLB, per-GPU private L2 TLB, and a shared IOMMU TLB. The baseline GPU TLB hierarchy employs the mostly-inclusive policy [19]. That is, when an IOMMU TLB miss occurs and the PTW is triggered, the requested translation is populated into IOMMU TLB, the L2 TLB and the L1 TLB. However,

whenever a translation is evicted from a lower level TLB, no invalidation is needed for the translation in the higher level TLBs.

Table 3: Single application workload.

Abbr.	Application	Benchmark Suite	MPKI
FIR	Finite Impulse Resp.	Hetero-Mark	0.009
KM	KMeans	Hetero-Mark	0.502
PR	PageRank	Hetero-Mark	0.409
AES	AES-256 Encryption	Hetero-Mark	0.003
MT	Matrix Transpose	AMDAPPSDK	2.394
MM	Matrix Multiplication	AMDAPPSDK	0.164
BS	Bitonic Sort	AMDAPPSDK	0.102
ST	Stencil 2D	SHOC	1.095
FFT	Fast Fourier Transform	SHOC	0.008

3.1.1.2 Workloads

We select nine applications from AMDAPPSDK [5], Hetero-Mark [71], and SHOC [23] benchmark suites. These applications are listed in Table 71. We use **workload** in this paper to represent a single application in single-application execution or multi-applications in multi-application execution.

Single-application workload: We first characterize single-application execution on multiple GPUs, i.e., "single-application-multi-GPU" execution paradigm. The applications cover different multi-GPU memory access patterns: random (BS, PR), adjacent (ST, FIR), partition (KM, AES), stride (FFT), and scatter-gather (MT, MM). To be more specific, workloads with random patterns exhibit random memory accesses from each GPU. The data sharing among GPUs is unpredictable. In contrast, the adjacent pattern shows overlapped memory footprint from neighboring GPUs. The partition pattern strictly partitions the data set among the GPUs and does not have any data sharing among GPUs, whereas the stride pattern shares data between different GPU pairs at each step. In the scatter-gather access pattern, each GPU reads/writes data from/to local memory and writes/reads data to/from the other GPUs, showing significant data sharing and forming a "producer-consumer" execution among the GPUs. The applications' memory footprints are sufficiently large to fill the TLB hierarchy in our targeted GPU architecture.

Multi-application workload: To study the "multi-application-multi-GPU" execution, we use applications shown in Table 71 and add another application – Simple Convolution

Table 4: Multi-application workload.

Abbr.	Workload	Applications	Category
W1	workload1	FIR, FFT, AES, SC	LLLL
W2	workload2	FIR, FFT, MM, KM	LLMM
W3	workload3	AES, SC, KM, PR	LLMM
W4	workload4	FFT, SC, KM, MT	LLMH
W5	workload5	AES, FIR, PR, ST	LLMH
W6	workload6	FIR, AES, MT, ST	LLHH
W7	workload7	FFT, SC, MT, ST	LLHH
W8	workload8	KM, PR, MM, BS	MMMM
W9	workload9	MM, KM, MT, ST	MMHH
W10	workload10	MT, MT, ST, ST	HHHH

(SC, from AMDAPPSDK with an MPKI of 0.018) to form multi-application workloads. Table 4 shows the ten workloads where each workload contains four applications[1]. The multi-application workloads are formed by characterizing their memory access intensity. Specifically, we quantify each application's misses-per-kilo-instructions (MPKI) of the address translations at L2 TLB. Based on the L2 TLB MPKI, we classify the applications into three categories: Low (L, MPKI<0.1), Medium (M, 0.1<MPKI<1), and High (H, MPKI>1). The ten workloads are formed as a mix of applications from different categories, including $LLLL$, $LLMM$, $LLMH$, $LLHH$, $MMMM$, $MMHH$, and $HHHH$. Note that, some applications may finish earlier in the concurrent execution. To maintain the TLB sharing and contention in multi-application execution, we adopt a similar approach from prior work [34, 85, 62]. That is, we ensure all GPUs are busy by re-executing applications that finish faster, until the longest-run application completes. Statistics are collected only for the first full execution of each application in a workload.

We use the following **metrics** in the paper:

- **Normalized performance**. Defined as the ratio of the execution time in the baseline approach to the execution time of our approach.
- **Reuse distance**. Defined as the number of unique translations between two accesses to the *same translation*. In multi-application environment, we calculate reuse distance considering the application process ID to differentiate the reuses from different applications with workload.
- **Weighted Speedup (WS)**. WS is used in the multi-application execution to give equal

[1]In our multi-application execution on four GPUs, each of the four applications occupies one GPU.

weight to the relative performance of each application [36]. WS is defined as the sum of each application speedup running in application mixes with respect to running alone. That is, $WS = \sum_{i=1}^{N} \frac{IPCapp_i(mix)}{IPCapp_i(alone)}$, where N is the number of applications in the workload.

3.1.2 Single-Application-Multi-GPU

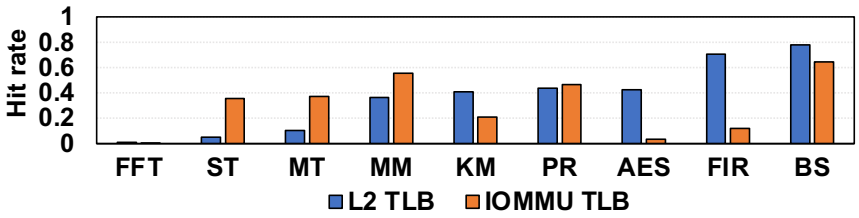

Figure 3: L2 TLB hit rate and IOMMU TLB hit rate in the baseline executions.

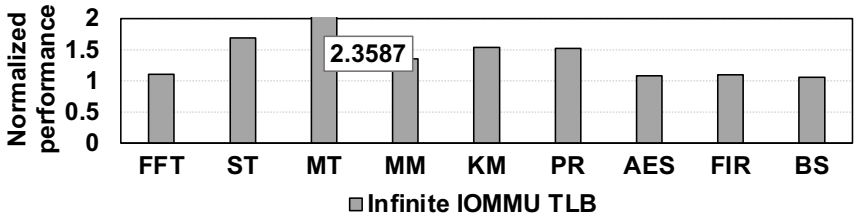

Figure 4: Normalized performance of infinite IOMMU TLB.

Observation 1: *Workload suffer from low TLB hit rate in both the L2 TLB and the shared IOMMU TLB.* Figure 3 shows the L2 TLB and IOMMU TLB hit rate in baseline execution. We observe low hit rates in both L2 TLB and IOMMU TLB. For example, `ST` with high MPKI has a hit rate of 5% in L2 TLB and 35% in IOMMU TLB. `AES` with low MPKI has a hit rate of 42% in L2 TLB and 3% in IOMMU TLB. The massive and intensive translation misses in the TLB hierarchy cause long latency in address translations.

To understand the impact of IOMMU TLB hit rate on performance, we measure the application performance under baseline and an infinite-sized IOMMU TLB (i.e., only cold misses exist). Figure 4 shows the performance normalized to the baseline execution. Overall, the infinite IOMMU TLB achieves 5.6% to 2.4x speedup, with an average performance improvement of 42.3%. We also observe that the improvement is more significant for applications with high MPKIs (e.g., `MT` and `ST`). As a result, there is great potential to improve performance by increasing the IOMMU TLB hit rate.

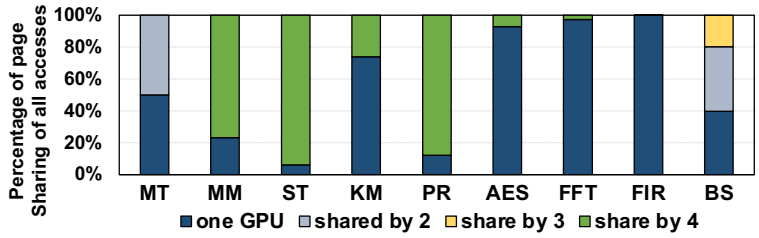

Figure 5: Percentage of page sharing.

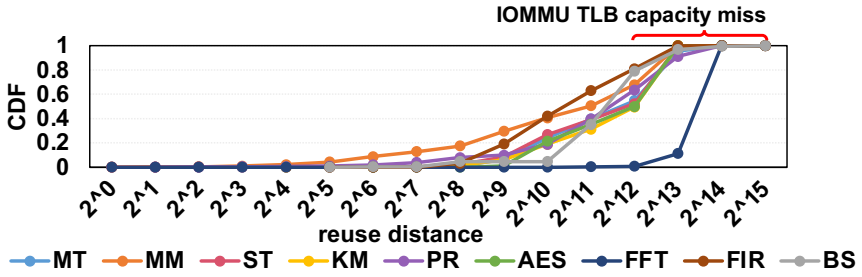

Figure 6: Cumulative distribution function (CDF) of translation reuse distances at the IOMMU TLB.

Observation 2: *A large fraction of translation reuses are not captured by the TLB hierarchy due to long translation reuse distances.* In the "single-application-multi-GPU" execution, different GPUs may access the same address translations during execution. To study the sharing behavior, we conduct a quantification of page reuses and plot the results in Figure 5. As shown in the figure, there exist a substantial fraction of address translations that are shared by multiple GPUs during execution. For example, in `MM`, more than 70% of the translations are shared by all four GPUs. In `PR` and `ST`, over 90% of the translations are shared. In `MT` and `BS`, about half of the translations shared between two or three GPUs. Figure 6 shows the cumulative distribution function (CDF) of the reuse distance for the address translation reuses in the IOMMU TLB. We also marked the IOMMU TLB capacity (4096) in the figure. On average, 45% of the reuses cannot be captured by the IOMMU TLB capacity.

Observation 3: *Translation reuses cause the exact translations to be redundantly stored in the TLB hierarchy, reducing the TLB reach.* Even if the reused translation can be captured by the TLB hierarchy, the same translation can be duplicated in both the GPU L2 TLB and

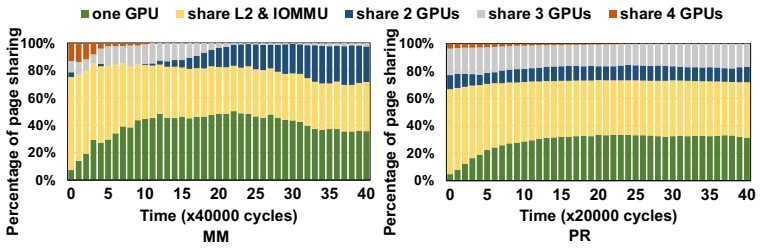

Figure 7: Page sharing during execution in MM and PR.

the IOMMU TLB in the baseline execution. In Figure 7, we take a snapshot of both the L2 TLBs' and the IOMMU TLB's contents at intervals of 40,000 cycles and 20,000 cycles for two workloads with high page sharing: MM and PR. One can observe that, the higher translation sharing, the more redundancy in the TLBs. On average, 25% and 30% of entries are stored in more than one GPU's TLB in the same cycle for MM and PR, respectively. Moreover, the same address translation may present in both the L2 TLB and the IOMMU TLB. For example, in MM, the percentages of entries stored in both the L2 TLBs and the IOMMU TLB range from 30% to 70%. As a result, this redundancy reduces the TLB reach (i.e., effective capacity), leading to more reuses to miss the TLB.

3.1.3 Multi-Application-Multi-GPU

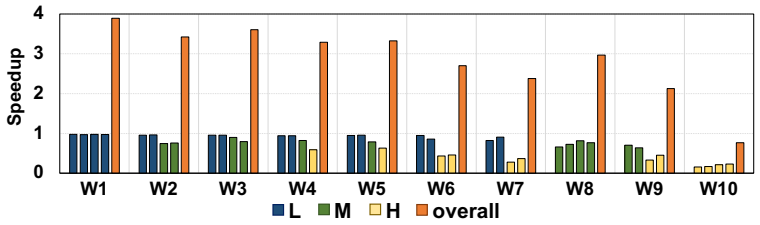

Figure 8: The speedup of each application in the workload and the overall weighted speedup of each workload.

In multi-application execution, application interference occurs in the shared IOMMU as each application exclusively runs on one GPU in the baseline. We quantify the performance impact caused by interference and contention at IOMMU TLB and Figure 8 shows the weighted speedup (defined in Section 4.1.1) of the ten workloads in Table 4. One can make the following observations. First, IOMMU TLB contention degrades individual application's

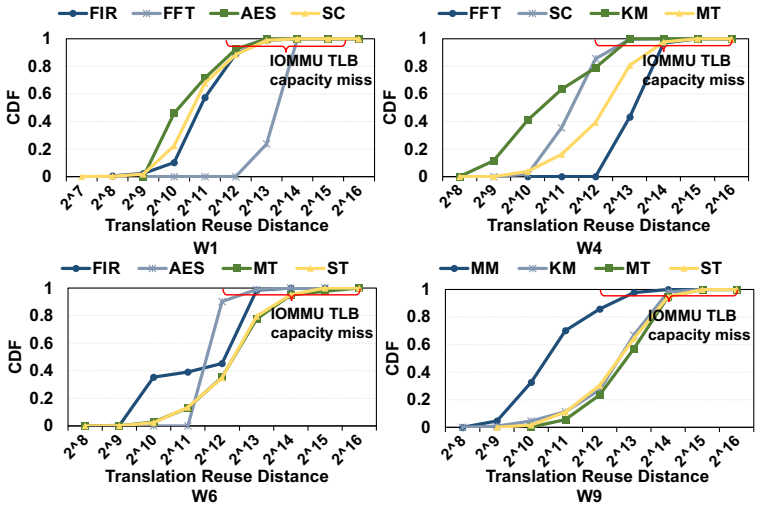

Figure 9: CDF of translation reuse distances in multi-application execution.

performance. In W1, the performance impact of each application is relatively minor. While in W10, performance drops by 77.5%. Second, the performance degradation of different applications in the same workload is different. For applications with higher MPKI, the impact is more significant. For example, in W6, the performance drop of `AES` (MPKI=0.003) is 15%, whereas the drop is 57% in `MT` (MPKI=2.394). This is because when the applications share the IOMMU TLB, translation requests often induce high contention among the high MPKI applications, so its TLB miss rate significantly increases. Third, the performance degradation of the same application in different workloads is different. For `MT`, the performance is reduced by 57% in W6 and 68% in W9. This is because, in W9, the co-running applications have a higher MPKI than those in W6, hence severer IOMMU TLB contention.

We further investigate the reuse distance of translations for multi-application execution. Figure 9 presents the translation reuse distances of four workloads with representative MPKI mix, i.e., LLLL, LLMH, LLHH, and MMHH. We observe that some applications (e.g., `FIR`, `AES`, and `KM`) have very different reuse distances in different workloads. For example, 89% of the reuse distance of `FIR` in W1 is less than the IOMMU TLB capacity (i.e., 4096), indicating a higher chance that these reuses can be captured in TLB. However, in W6, only 45% of the reuses in `FIR` are within the TLB capacity. This is because in W6, `ST` and `MT` have high MPKIs, and they generate a large number of translation requests to IOMMU. Therefore,

the reuse distance of FIR is extended because of contention. For applications like ST and MT, they generate intensive translation requests that miss L2 TLB and occupy a significant portion of entries in IOMMU TLB. Therefore, their reuse distances does not change much in each workload. We also marked the IOMMU TLB capacity in the figure. As one can observe, for severe contention applications(i.e., MT and ST), more than 60% of the reuses are missed in the IOMMU TLB.

3.2 Least-TLB Design

Goals: Our goal in this paper is to improve the multi-GPU TLB hit rates, thereby boosting the performance of both single-application execution and multi-application execution. To this end, we develop the least-inclusive TLB (also called *least-TLB*) that takes advantage of translation sharing (in single-application) and spilling (in multi-application) to reduce translation redundancy and mitigate the contention in IOMMU TLB.

Challenges: Designing least-TLB faces several challenges. First, in a least-inclusive TLB hierarchy, not all L2 TLB entries are present in the IOMMU TLB. So one GPU might find the desired address translation in another GPU's L2 TLB rather than in the IOMMU TLB. In such a scenario, querying only the IOMMU TLB will result in a miss, which can be avoided. Second, in multi-application-multi-GPU execution, it is important to select an appropriate GPU's L2 TLB as the receiver for IOMMU TLB spilling. Receiving spilled entries may introduce contention in that GPU's L2 TLB. In addition, the receiver GPU should be selected dynamically during execution by considering the phase behavior of applications where the translation request intensity may vary. Finally, the hardware overhead of least-TLB should be minimized to make it feasible in the practical GPU hardware.

3.2.1 Single-Application-Multi-GPU

In this paper, we propose *least-TLB* design where the IOMMU TLB is used as a "victim TLB" for the GPU L2 TLBs. That is, translations are only inserted into the L2 TLB upon

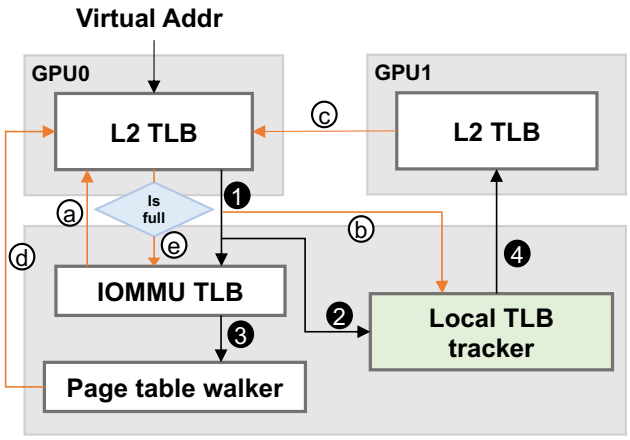

Figure 10: The lookup and insertion in least-TLB for single-application execution.

lookup, and only when the translations are evicted from the L2 TLB, they will be placed into the IOMMU TLB. Note that, our design does not affect the GPU L1 TLB and L2 TLB where the mostly-inclusive policy is used. The key hardware structure used in least-TLB is the Local TLB tracker (as highlighted in Figure 10) in IOMMU.

Local TLB tracker: Although the proposed least-inclusive TLB design reduces translation redundancy, it causes extra IOMMU TLB misses compared to the inclusive and most-inclusive designs. This is because, least-inclusive policy brings a translation directly to the L2 TLB without allocating an entry in the IOMMU TLB. When another GPU tries to access the same translation, it cannot find the translation in the IOMMU TLB and a page table walk request is issued. To allow sharing while maintaining the proposed least-inclusive policy, we implement the Cuckoo filter[27] in the IOMMU to *track* the translations in all GPUs' L2 TLBs. At a high-level, Cuckoo filter is similar to Bloom filters [20], and is a space-efficient data structure that tests whether an element is in a set. It uses hash functions to derive the inserted items into a bit string of fingerprints. Each inserted item is stored as a fingerprint instead of a key-value pair. The Cuckoo filter supports efficient deletion operations. It checks two candidate buckets for a given item; if any fingerprint in any bucket matches, it deletes a copy of the matching fingerprint from that bucket. If two items share the same bucket and fingerprint, a random one is selected and deleted, causing false positive cases. Another source of false positive is the repetition of fingerprints. That is, multiple elements may produce the same fingerprint. In the context of TLB tracking, when a translation is brought to the L2

Algorithm 1: Single-application Lookup & Insertion.

```
1  /* Lookup () */
2  if hit in L2_TLB then
3      update L2_TLB;
4      respond to L1 TLB and L1$;
5  else
6      Pthread IOMMUTLB and Tracker;
7      if hit in IOMMU_TLB then
8          end Pthread_Tracker;
9          Insert()→L2_TLB;
10         evict translation from IOMMU_TLB;
11         add translation to Tracker;
12     else
13         PTW;
14         Insert()→L2_TLB;
15     if positive in Tracker_x then
16         forward request to GPU_x;
17         if hit in GPU_x then
18             end Pthread_IOMMUTLB;
19             Insert()→L2_TLB;
20             add translation to Tracker;
21     respond to L1 TLB and L1$;
22
23 /* Insert () */
24 if L2_TLB_i is full then
25     insert the first translation of LRU list into IOMMU_TLB;
26     delete translation from Tracker_i;
27 if IOMMU_TLB is full then
28     evict the first translation of LRU list ;
```

TLB, it is also registered in the Cuckoo filter hardware. When a translation is evicted from the L2 TLB and inserted into the IOMMU TLB, it is also removed from the Cuckoo filter.

TLB lookup: Figure 10 shows TLB lookup procedure in least-TLB during single-application execution. The corresponding algorithm is given in Algorithm 1. Specifically, when a translation request arrives at the IOMMU, the IOMMU TLB (❶) and the Local TLB tracker (❷) are searched in parallel. Depending on where the translation is present, three different scenarios may occur. First, if the request hits in the IOMMU TLB, the translation is fetched to L2 TLB and the lookup in the Local TLB tracker is abandoned (ⓐ). This fetched translation is also placed in the corresponding Local TLB tracker (ⓑ) for future references and is removed from the IOMMU TLB based on the proposed least-inclusive policy (lines 7 to 11). Second, if the request misses the IOMMU TLB and hits the tracker, the request is forwarded to the corresponding remote GPU and is served by the remote GPU's L2 TLB (❹). Considering the shared translation may be accessed frequently, we keep the translation in both the sender's L2 TLB and the receiver's L2 TLB (ⓒ) and update the tracker in the IOMMU (ⓑ) (lines 16 to 20). Note that, it can happen that the Cuckoo filter provides a

false prediction and the remote GPU does not hold the translation. In such a case, the IOMMU still sends the miss requests to the PTWs after IOMMU TLB lookup (❸), hiding the latency caused by the tracker mis-prediction (lines 12-13). Note also that, retrieving a translation from page table could be faster than accessing remote TLB in some cases (e.g., the interconnect is congested). Therefore, performing PTW and remote lookup simultaneously can avoid causing additional latencies in such cases. In our implementation, the IOMMU uses a lookup table to track the pending requests that are sent to PTW and the remote TLB. Whichever comes first, the table translation is served to the requesting GPU and the table is updated by removing the request. When the same translation comes again, it will be discarded as the request has been served already. Third, if the request misses both IOMMU TLB and the Local TLB tracker, it is sent to the PTWs. When the request returns, the translation is only inserted in the L2 TLB (ⓓ) (line 14) due to least-inclusive policy.

TLB insertion: When the L2 TLB is full, one entry is evicted from the L2 TLB based on LRU policy. The evicted entry is placed in the IOMMU TLB (ⓔ) (lines 24 to 26) and the translation is also removed from the Local TLB tracker.

Note that, our proposed least-TLB is *not* equivalent to an exclusive TLB hierarchy. An exclusive hierarchy guarantees that one translation is presented in either one of GPUs' L2 TLBs or the IOMMU TLB, but not both. In our least-inclusive TLB hierarchy, when a translation is evicted from one GPU's L2 TLB, if it is inserted into the IOMMU TLB, we do not invalidate the translation in other GPUs' L2 TLBs. As a result, a translation may exist in both the GPU's L2 TLB and IOMMU TLB at the same time.

3.2.2 Multi-Application-Multi-GPU

Now let us discuss how we leverage the proposed least-TLB hierarchy in the "multi-application-multi-GPU" execution. Recall our discussion in Section 4.1.3, when multiple applications run concurrently on multiple GPUs, they compete for the IOMMU TLB, leading to increased IOMMU TLB misses and performance degradation. To mitigate the contention, our intuition is to leverage the GPUs' L2 TLB as a temporary "victim buffer" for the entries that are evicted from the IOMMU TLB. However, as discussed earlier, spilling IOMMU

TLB entries to a GPU's L2 TLB may slow down the local application execution due to extra L2 TLB thrashing. Fortunately, we observe that some compute-intensive applications are less sensitive to TLB miss, compared with memory-intensive applications. As a result, when the co-running applications have mixed MPKIs, those applications with low MPKI could be suitable candidates for receiving the TLB spilling translations. Based on the above observation, we answer the following questions in multi-application execution regarding how the proposed least-TLB can be used: *what to spill?*, *where to spill?*, and *how to spill?*.

What to spill: In multi-application execution, we allow the evictions from the IOMMU TLB to have more chances to reside in the TLB hierarchy by spilling them to the other GPU's L2 TLB when possible. This is extremely helpful to capture long reuses distances caused by interference from concurrent executions and improves the weighted performance when the GPU receiver executes an application that is insensitive to its L2 TLB performance. Potentially, one can allow the spilling chances of each entry by specifying a counter N. When N equals 1, each translation is associated with one extra spilling bit which is initialized to 1. The bit is set to 0 when the translation is evicted from IOMMU TLB and is spilled to another GPU's L2 TLB. Later, when the same entry is evicted from the L2 TLB, the spilling bit is checked and the entry is abandoned without putting it in the IOMMU TLB due to least-inclusiveness. If an access/reuse happens to a spilled entry, the tracker informs the requesting GPU of the location of the entry, and the spilled bit is reset to 1. Note that, larger N gives the translation more opportunities to recirculate through the TLB hierarchy and is, therefore, more likely to capture long-distanced reuses. However, when both the IOMMU TLB and the L2 TLB are full, the spilling between the L2 TLBs and IOMMU TLB may cause a "chain" effect where ping-pong evictions can happen. With a large N, there expect a severe chain effect. Therefore, in our design, we set $N = 1$. We also provide sensitivity study in Section 3.3.3 with different values of N.

Where to spill: Next, it is important to determine which GPU should receive the spilled translations. Ideally, we want to choose the GPU whose L2 TLB is least thrashed, and whose running application is insensitive to L2 TLB performance. Meanwhile, in many applications, the TLB access intensity varies during program execution. We need to choose the receiver GPU dynamically by considering the TLB access intensity in different execution

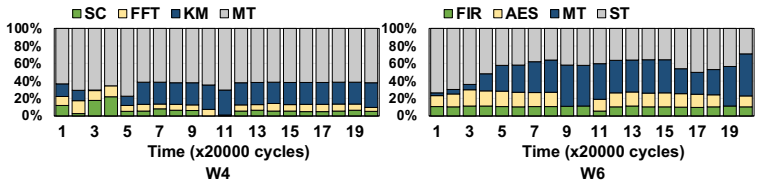

Figure 11: IOMMU TLB contents during executions of W4 and W6.

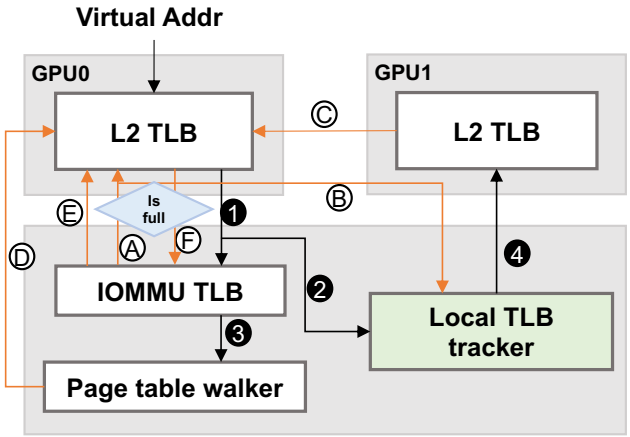

Figure 12: The lookup and insertion in least-TLB for multi-application execution.

phases.

To this end, in our least-TLB design, each GPU can potentially act as a receiver of the IOMMU evictions. To help find the most suitable candidate GPU receiver, we choose two representative workloads with hybrid MPKIs and take snapshots of the IOMMU TLB contents at specific intervals, as shown in Figure 11. One can observe that for those applications with higher L2 TLB thrashing, more translations are kept in the IOMMU TLB. Therefore, we use the number of translations present in the IOMMU TLB to determine the receiver GPU dynamically. We introduce a hardware *Eviction Counter* for each GPU in the IOMMU to record the IOMMU TLB entries from different GPUs. Whenever an IOMMU eviction happens with the spill bit set to 1, we use the Eviction Counter to select the GPU receiver that has the smallest counter value.

How to spill: We use the same Cuckoo filter based Local TLB tracker to track the spills in multi-application execution. When a translation in the IOMMU TLB spills to a GPU's L2 TLB, it is recorded in the Local TLB tracker. When the spilled translation is evicted from the receiver GPU or hit by the original GPU again, it is removed from the receiver's

Algorithm 2: Multi-applications Lookup & Insertion.

```
/* Lookup () */
if hit in L2_TLB then
    update L2_TLB;
    respond to L1 TLB and L1$;
else
    Pthread IOMMUTLB and Tracker;
    if hit in IOMMU_TLB then
        end Pthread_Tracker;
        Insert()→L2_TLB;
        evict translation from IOMMU_TLB;
    else
        PTW;
        Insert()→L2_TLB;
    if positive in Tracker_x then
        forward request to GPU_x;
        if hit in GPU_x then
            end Pthread_IOMMUTLB;
            Insert()→L2_TLB;
            delete translation from Tracker_x;
    respond to L1 TLB and L1$;

/* Insert () */
if L2_TLB_i is full then
    if the spill bit of first translation of LRU list == 1 then
        insert translation into IOMMU_TLB;
        Eviction_Counter_i ++ ;
    else
        evict translation;
        delete translation from Tacker_i;
if IOMMU_TLB is full then
    min = min(Eviction_Counter) ;
    insert the first translation of LRU list to GPU_min;
    Eviction_Counter −− ;
    add translation to Tacker_min ;
```

L2 TLB and is removed from the tracker. The Cuckoo filter configuration is the same as the single-application execution.

Figure 12 shows the lookup and insertion procedures. The algorithms are given in Algorithm 2.

TLB lookup: The lookup process is similar to the lookup in single-application execution. The only difference is that when a translation hits in the other GPU's L2 TLB (i.e., remote hit), unlike in single-application execution where it presents in both the requesting and receiver GPUs, the translation is removed from the receiver GPU's L2 TLB in multi-application execution. That is because there is no translation sharing among the applications running on different GPUs. Therefore, this is no need to keep the spilled translation in the GPU receiver after the original GPU requests it. After fetching, the Local TLB tracker in the IOMMU is updated accordingly.

TLB insertion: When an eviction occurs from the GPU's L2 TLB, we first check the spill bit of the eviction. If the bit is 1 (meaning it is not a spilled translation), the translation is inserted into the IOMMU TLB (Ⓕ), and the Eviction Counter is updated (lines 24 to 26). Otherwise, when the spill bit is 0 (meaning it is a spilled entry), we simply discard the translation and delete the record from the Local TLB tracker (lines 28 to 29). When an IOMMU eviction occurs, the evicted translation is inserted into the GPU receiver's L2 TLB (Ⓐ). As we discussed before, the receiver is selected as the GPU that has the fewest entries in the IOMMU TLB. The spill bit is also set to 0, and the Eviction Counter is updated. The translation is registered in the Local TLB tracker (Ⓑ), so that future accesses to the same address can query the tracker (lines 31 to 34).

3.2.3 Hardware Overhead

In our configuration, the Cuckoo filter has a total of 2048 entry with 0.2 false positive probability. The Cuckoo filter are divided equally according to the number of GPUs. The total hardware overhead of Cuckoo filter is 1.08KB. Our design also requires 32-bit for four Eviction Counter to record the number of entries stored in the IOMMU TLB. We use CACTI [76] to estimate the area and power overheads of our approach. The result shows 0.19% area overhead compared to the area of IOMMU TLB.

3.3 Evaluation

In this section, we evaluate the proposed least-TLB design using MGPUSim [72]. The system configuration is the same as the baseline shown in Table 5.

3.3.1 Single-Application-Multi-GPU Execution

Figure 13 shows the performance improvements brought by least-TLB and an impractical IOMMU TLB design with infinite entries, both normalized to the baseline. The proposed least-TLB delivers an average speedup of 1.24× over the baseline. Specifically, the five

applications (ST, MT, MM, KM, PR) achieve an average performance speedup of 1.38×. This is because these applications have either medium (M) or high (H) MPKI values (in Table 71) and benefit more from the least-TLB design. The remaining four applications have relatively low (L) MPKI values, so the performance improvements are less. Overall, least-TLB achieves comparable performance improvement compared with the infinite IOMMU TLB, with the exception of MT. For MT, the reason is that the reuse distance is too large to be captured by the limited TLB capacity even after reducing redundancy.

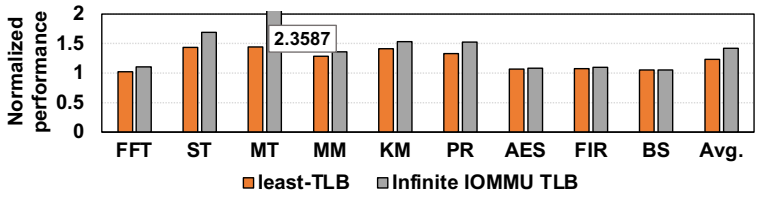

Figure 13: Normalized performance improvements in single-application execution.

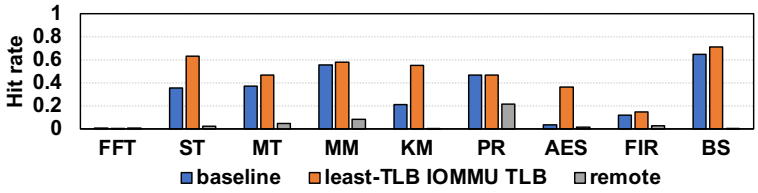

Figure 14: IOMMU TLB hit rate and remote hit rate in single-application execution.

We further look into the IOMMU TLB hit rate and remote GPU L2 TLB hit rate to understand the reason behinds the performance improvements. As shown in Figure 14, least-TLB improves the IOMMU TLB hit rate by 12.9% across all applications, and the average remote L2 TLB hit rate is 4.7%. Specifically, in workloads like ST, MT, MM, KM, and PR, where the degree of page sharing is relative high (as illustrated before in Figure 5), least-TLB improves the hit rate by 22.2% on average (including both IOMMU and remote L2 TLBs). This is because, least-TLB reduces duplicated address translations in IOMMU and L2 TLBs, which in turn improves the TLB reach, allowing more translation reuses to be captured. Although the amount of translation sharing among GPUs is low in AES, there is still a significant improvement in its IOMMU TLB hit rate due to the increased TLB reach. For FIR, BS, and FFT, the impact of minimizing redundant translations is limited and the reuse distances of FIR and BS are mostly within baseline IOMMU TLB capacity. Therefore, the improvement on hit rate is minor. However, we want to emphasize that least-TLB does

not incur any extra misses. In other words, it does not hurt the application performance that is already good in the baseline execution.

3.3.2 Multi-Application-Multi-GPU Execution

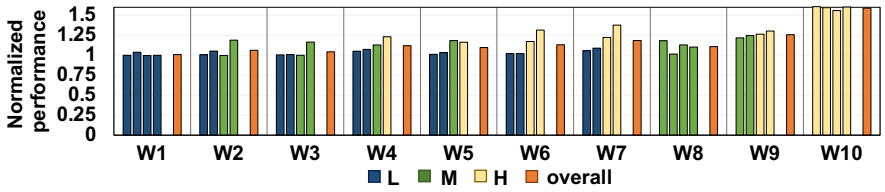

Figure 15: Normalized performance improvements in multi-application execution.

Figure 15 shows the weighted performance improvements (denoted by the last bar of each workload) of the multi-application workloads listed in Table 4. For a clear comparison, we also show the performance improvements of individual applications within each workload. Results are normalized to the baseline multi-application execution. From the figure, one can make the following observations. First, the proposed least-TLB improves the performance up to 59.1%, with an average of 16.3% across all workloads. The improvement is more significant for workloads that suffer from severe contention in the IOMMU TLB. For example, W4 ($LLMH$) achieves 12% improvement and W7 ($LLHH$) achieves 18% improvement, respectively. This is because workloads that comprise applications with mixed MPKIs are likely to find a GPU candidate running a low MPKI application (i.e., with less thrashing in its L2 TLB) to act as a receiver for the translation spilling from the GPU that runs high MPKI application. One interesting observation is that, for W10, all four applications have high MPKI and W10 still achieves significant improvement. This is because these applications show interleaved intensity in TLB access patterns, i.e., some applications access TLBs more intensively during execution periods that others are not. As such, our approach dynamically selects the receiver GPUs in different phases. Second, within the same workload, the performance improvement is larger for applications with high MPKI, indicated by the yellow bars in the figure. The improvement mainly comes from the increased hit rate (including IOMMU TLB and remote L2 TLB) in least-TLB design. Finally, for applications with low MPKIs within a workload, our least-TLB does not provide as much improvement as we achieved for applications with high MPKIs. This is due to the fact that either these applications are TLB

insensitive, or the baseline TLB is already doing a good job in capturing all the translation reuses.

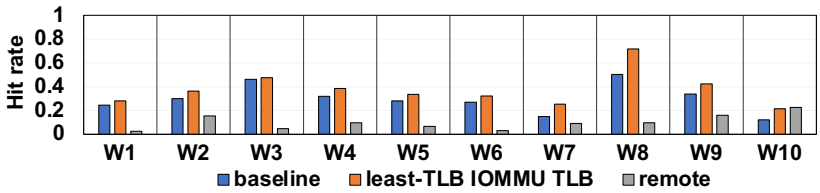

Figure 16: IOMMU TLB hit rate and remote hit rate in multi-application execution.

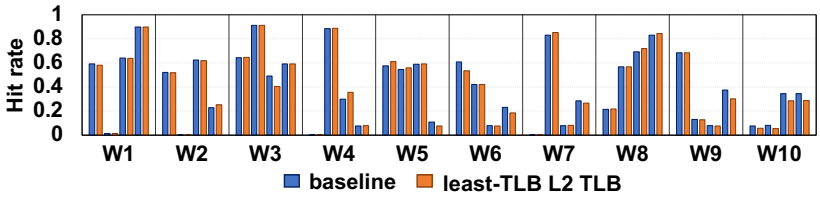

Figure 17: L2 hit rate in multi-application execution.

Figure 16 and Figure 17 plot the IOMMU TLB hit rate and L2 TLB hit rate for each application in each workload. We also show the remote hit rate in Figure 16. First, on average, least-TLB improves IOMMU TLB hit rate by 7.8% across all workloads and achieves an average remote hit rate of 10%. The increase in TLB hit rate directly translates to performance improvement. Comparing the remote hit rate with the IOMMU TLB hit rate improvement (difference between orange and blue bars in Figure 16), we notice that in most workloads, the remote hit rate is larger. This indicates that our least-TLB effectively captures long-distance translation reuses by allowing dynamic spilling. For example, the average IOMMU hit rate of W10 is 21.5% that is 9.3% better than the baseline, whereas the average remote hit rate is 22.5%. Second, the L2 TLB hit rate is not affected in most applications. On average, the L2 hit rate of least-TLB is 3% lower than the baseline. However, the hit rate drop is more obvious in W10. This is because `ST` and `MT` have H MPKI, which are sensitive to TLB misses. Spilling IOMMU TLB entries to L2 TLBs can cause L2 TLB thrashing. Finally, in most workloads, applications with higher MPKIs have higher hit rate improvement. For example, in W2, W6 and W7, `KM` and `ST` have high MPKI in the corresponding workloads and their hit rate improvement is higher compared to other applicants in the same workloads. However, for `KM` in W3, the observation is the opposite. This is because, although applications with H MPKIs generate a number of spilled translations in the runtime, these

spilled translations are evicted before they can be reused. This might happen if the GPU receiver changes the execution phase and becomes TLB intensive. As a result, the spilled translations will be evicted from the receiver based on the LRU replacement policy.

3.3.3 Sensitivity Study

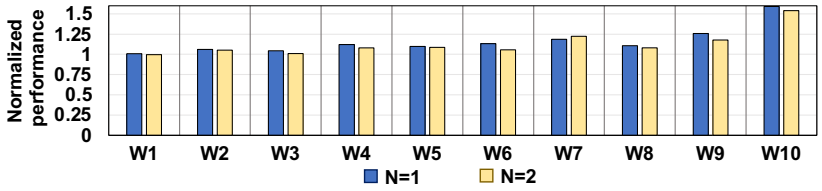

Figure 18: Normalized performance of spilling counter $N = 2$.

Sensitivity to spilling counter: In our discussion so far, the spilling counter N is set to 1, indicating that the IOMMU TLB evicted entry only gets one chance to spill to other GPUs' L2 TLBs. In this study, we evaluate the performance impact when changing the spilling counter to *N=2*. Figure 18 shows the results. While the average performance improvement is 12.7% over the baseline, there is 3.1% performance drop compared to *N=1*. The main reason behind this is the "chain" effect, which causes ping-pong eviction between L2 TLBs and the IOMMU TLB.

Sensitivity to the IOMMU TLB size: We evaluate least-TLB under a smaller IOMMU with 2048 entries [30]. The results show an average performance improvement of 14.7% and 10.2% in single-application and multi-application executions, respectively. The performance benefits are lower (as opposed to 23.5% and 16.3% when using a 4096-entry IOMMU TLB). This is because a smaller IOMMU TLB will reduce the chances of reuses being captured.

Sensitivity to the remote access latency: We evaluate least-TLB under different remote GPU TLB access latencies to show the cross-over point when accessing a remote GPU compared against invoking the page table walk in the DRAM. Figure 19 shows the scaling results. The black solid line represents the baseline mostly-inclusive implementation. As none of the requests is going to the remote GPUs, the performance does not vary when remote GPU access latency changes. The colored solid line represents the results where all the requests indicated as positive by the Cuckoo filter are sent to remote GPUs, and only those missing in the remote GPU TLB will access the page table in the DRAM. One can

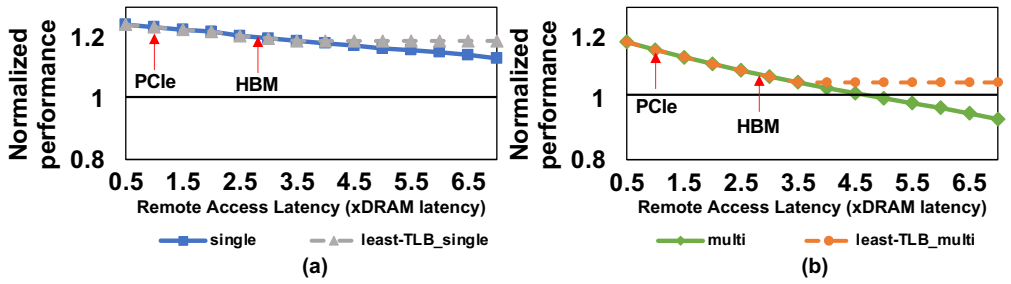

Figure 19: Normalized performance with different remote GPU access latencies. (a) single-application. (b) multi-application.

observe that, for example in multi-application execution, when the remote access latency is beyond 5× of the DRAM latency, the performance of accessing the remote GPU is worse compared to accessing the page tables in the DRAM. The dashed lines in Figure 19 represent the results of least-TLB. As the PTW and the remote TLB lookup are performed simultaneously, the translation is served by whichever comes first. Therefore, after the remote accessing latency is higher than the average PTW latency (3.5× of DRAM latency in multi-application execution), least-TLB can maintain performance benefits compared to waiting for remote lookups. Note that after 3.5×, our least-TLB still performs better than the baseline where all the translations missed on the IOMMU TLB are served by the PTW. This is due to the increased IOMMU TLB hit rate brought by the least-inclusive policy in least-TLB. We also marked the PCIe and HBM latencies in the Figure 19. The CPU and GPUs are connected via PCIe (˜300ns latency) in our configuration, and the latency of HBM (˜106.7ns latency) is much less than that of PCIe. It is important to note that, as the DRAM technologies evolving, the DRAM latency is reducing and the memory bandwidth is increasing (e.g., HBM [70] and HBM2 [35]). On the contrary, the interconnection can be congested when multiple devices, especially heterogeneous ones with different quality of service (QoS) requirements, are connected to the IOMMU. As a result, multi-application execution may prefer invoking DRAM page table walk instead of going remote devices. However, for single application execution, the substantial address translation sharing may still prefer remote TLB accesses, as indicated in Figure 19 where the cross-over point has higher remote latency in single-application execution.

Sensitivity to the number of GPUs: We evaluate least-TLB with 8 GPUs and 16

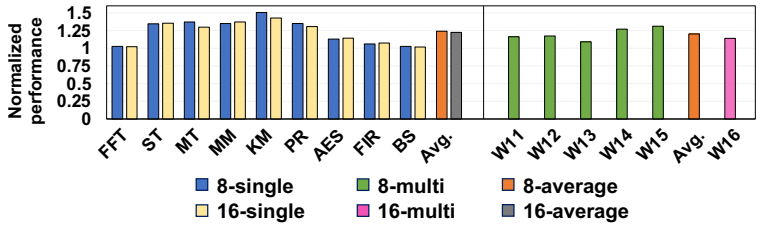

Figure 20: Normalized performance of least-TLB with 8 and 16 GPUs.

GPUs to show its scalability. Figure 20 plots the performance of single-application execution normalized to baseline execution. The average performance improvement of 8-GPU and 16-GPU is 24.1% and 22.5%, respectively. Figure 20 also shows the multi-application execution results. We use applications in Table 71 to form 5 8-GPU workloads and a 16-GPU workload. We observe that the performance of 8-GPU achieves an average of 20.2% improvement and 16-GPU achieves 14.0%. In a nutshell, the proposed least-TLB is able to deliver scalable performance improvements with more GPUs.

3.3.4 Comparison to Large-sized Pages

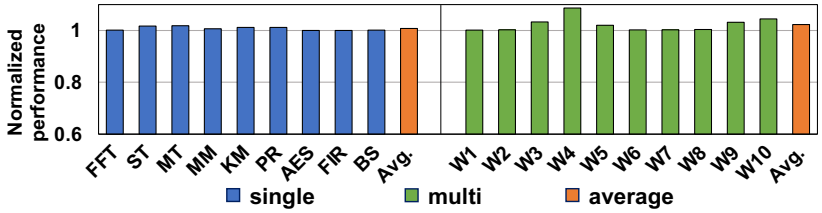

Figure 21: Normalized performance when using with 2MB page.

Regardless of the deficiencies in large pages, e.g., fragmentation, large pages generally effectively improve the TLB reach. In this part, we evaluate how least-TLB works with large-sized pages (i.e., 2MB). Figure 21 presents the normalized performance when we apply least-TLB on 2MB page size. The results are normalized to baseline 2MB page execution. We observed the average speedups over the baseline are 0.78% in single-application execution and 2.3% in multi-application execution, respectively. While it is expected that the improvements are less since large pages intuitively improve the TLB reach, our proposed least-TLB is able to further improve the overall performance when combined with large pages, especially for multi-application execution that has diverse TLB access patterns from different applications.

3.3.5 Comparison to State-of-the-art

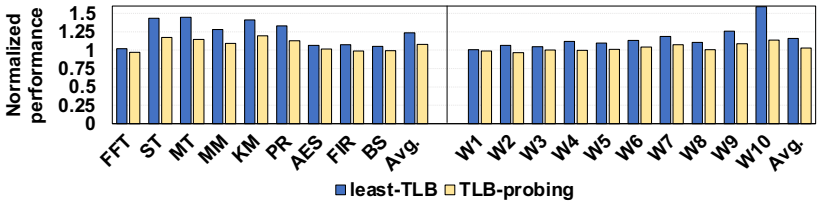

Figure 22: Comparison to TLB probing [12].

We compare least-TLB with the state-of-the-art TLB probing scheme [12]. The original work [12] focused on a single-GPU system, and TLB-probing scheme was proposed to enable translation sharing among L1 TLBs. In our comparison, we extend TLB-probing scheme to L2 TLBs and connect all L2 TLBs from all GPUs into a ring network, such that TLB probing requests can be sent between any neighboring GPUs. Figure 22 shows the performance comparison for single-application and multi-application. On average, our approach outperforms TLB-probing by 15.7% in single-application execution and 13.1% in multi-application execution. The main reason is that, when a GPU L2 TLB miss occurs, TLB-probing scheme sends two requests to neighbor GPUs for lookup. Such a ring-based lookup works well within single GPU (as the target in [12]). It is less efficient in the multi-GPU scenario due to long probing delay and low TLB reaching. In contrast, our least-TLB avoids such remote queries by lookup in the Local TLB Tracker.

3.3.6 Combined with PTW optimization

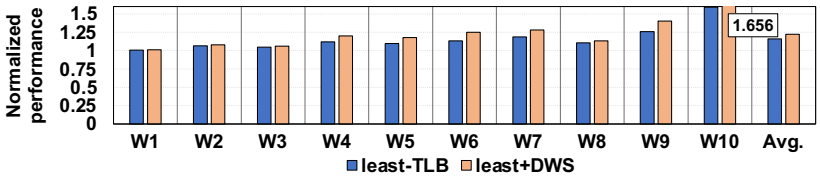

Figure 23: Combined with DWS [62].

To show the proposed least-TLB can achieve better performance when combining with prior PTW optimizations, we apply least-TLB to one state-of-the-art PTW optimization proposed by Pratheek et al. [62]. They focused on mitigating PTW contention under multi-tenancy on a single GPU. Figure 23 shows that the least-TLB+DWS achieves a 22.4% performance improvement in multi-application execution, which is a 6.1% improvement over the

least-TLB alone. In short, we demonstrate that least-TLB can work with PTW optimization and bring additional performance benefits.

4.0 Trans-FW: Short Circuiting Page Table Walk in Multi-GPU Systems via Remote Forwarding

4.1 Motivation

4.1.1 Baseline Configuration and Applications

We conduct our characterization and later evaluate our proposed Tran-FW using the MGPUSim [72]. To model unified virtual memory and the full address translation process, we substantially modified and extended MGPUSim by adding i) per-GPU GMMU module with GPU local page tables, local PW-queue, and local PW-cache, and ii) host MMU module with a host TLB, host MMU page table, host MMU PW-cache, and host MMU PW-queue.

Table 5: Baseline multi-GPU configuration.

Module	**Configuration**
CU	1.0 GHz, 64 per GPU
L1 Vector Cache	16 KB, 4-way
L1 Inst Cache	32 KB, 4-way
L1 Scalar Cache	16 KB, 4-way
L2 Cache	256 KB, 16-way
DRAM	4 GB
L1 TLB	32 entries, 32-way, 1-cycle lookup latency, CU private, LRU replacement policy
L2 TLB	512 entries, 16-way, 10-cycle lookup latency, CUs shared, LRU replacement policy
Host MMU TLB	2048 entries, 64-way, GPUs shared, LRU replacement policy
Page table walk	Host MMU 16 shared page table walker, GMMU 8 shared page table walker [67, 79, 62], 100-cycle latency per level [30]
Page walk cache	128 entries shared across page table walker [62]
Page walk queue	64 entries
CPU-GPU interconnection	PCIe, 150-cycle latency [30]

Baseline GPU configuration: In this paper, we target a 4-GPU system where each GPU has its own page table stored in its device memory [81, 42, 41]. The detailed baseline configurations are listed in Table 5. Note that, our approach is also applicable to different GPU counts and we provide a sensitivity study with 8 and 16 GPUs in Section 4.3.2. We employ a five-level nested page table organization, thereby a four-level PW-cache [31]. We use UTC PW-cache organization. The total size of the PW-cache is typically in the range

Table 6: List of applications.

Abbr.	Application	Benchmark Suite	PFPKI	Access Pattern
AES	AES-256 Encryption	Hetero-Mark	0.016	Partition
FIR	Finite Impulse Resp.	Hetero-Mark	0.002	Adjacent
KM	KMeans	Hetero-Mark	3.636	Adjacent
PR	PageRank	Hetero-Mark	9.244	Random
MM	Matrix Multiplication	AMDAPPSDK	3.217	Scatter-Gather
MT	Matrix Transpose	AMDAPPSDK	34.273	Scatter-Gather
SC	Simple Convolution	AMDAPPSDK	9.013	Adjacent
ST	Stencil 2D	SHOC	17.564	Adjacent
Conv2d	Convolution 2D	DNN-Mark	1.782	Adjacent
Im2col	Image to Column	DNN-Mark	1.198	Scatter-Gather

of 64 to 128 [10, 62, 43] under a four-level page table. We employ 128-entry PW-cache in our five-level page table in GMMU and host MMU. The CTA policy in the baseline and our approach is as follows. The CTA scheduler first schedules the CTAs across CUs within a GPU in a round-robin fashion, and then moves to the next GPU only when the GPU has no available resources. That is, the CTA is scheduled greedily across GPUs. This scheduling captures the inter-CTA locality within a GPU and also maintains computing balancing across CUs.

Applications: We use 10 applications from Hetero-Mark [71], AMDAPPSDK [5], SHOC [23], and DNN Mark [25] benchmark suites. The details of the applications are listed in Table 71. We use their multi-GPU implementation from [72], which is also used by prior works [40, 73, 11]. The applications cover a wide range of data access/sharing patterns in multi-GPU environment. We classify these applications into four categories based on their memory access patterns: random (PR), partition (AES), adjacent (ST, FIR, SC, Conv2d, KM), and scatter-gather (MT, MM, Im2col).

4.1.2 GPU Page Walk Characterization

Figure 24 shows the latency breakdown of GPU L2 TLB misses. One can make the following observations. First, handling local page faults causes significant overhead and accounts for 86.1% of the L2 TLB miss latency. The local page faults latency can be further breakdown into i) waiting in the shared host MMU PW-queue, ii) missing the host MMU PW-cache, iii) migrating page to local memory, and iv) CPU-GPU interconnection and request replayed when page fault is resolved. Second, 25.0% of the latency is caused by

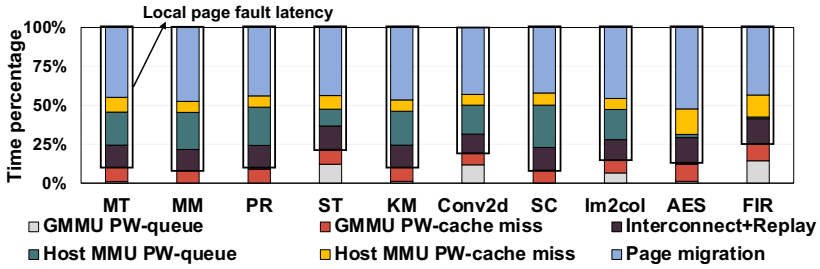

Figure 24: Latency breakdown of GPU L2 TLB misses.

requests waiting for the available page table walk thread in the PW-queue. The PW-queue queuing latency in shared host MMU is generally longer than that in the GMMU for most applications. Specifically, the average queuing latencies are 4.1% and 20.9% for the GMMU PW-queue and the host MMU PW-queue, respectively. This is because the host MMU handles page faults generated from all GPUs, encountering more severe contention than local GPUs. Finally, an average of 9.0% and 9.3% latency is caused by GMMU and host MMU PW-cache miss, respectively.

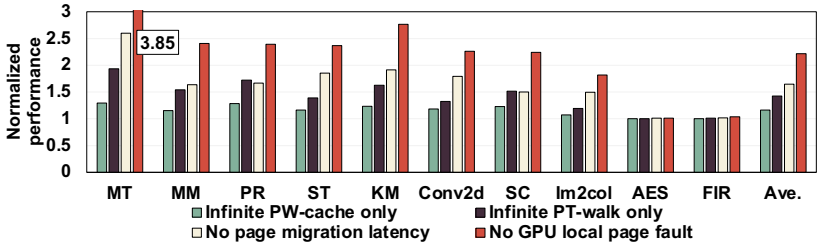

Figure 25: Performance improvements when latency sources are resolved.

Room for improvement: We study the performance gains when we i) adopt infinite PW-cache in both GPU and host MMU, ii) employ infinite page table walking threads in both GPU and host MMU, and iii) eliminate all GPU local page faults. The normalized execution performance is shown in Figure 25. In the results, we separate the influence of each latency by applying only one of the above three impractical optimizations at a time. For example, when we adopt infinite PW-cache, the other two configurations remain the same as in the baseline. We next discuss the implication from each individual optimization in detail.

Implication of infinite PW-cache: An infinite-sized PW-cache only has cold misses. We employ infinite-sized PW-cache in both GMMU and host MMU. The first bar of each application in Figure 25 shows the performance improvement of an infinite PW-cache. Overall,

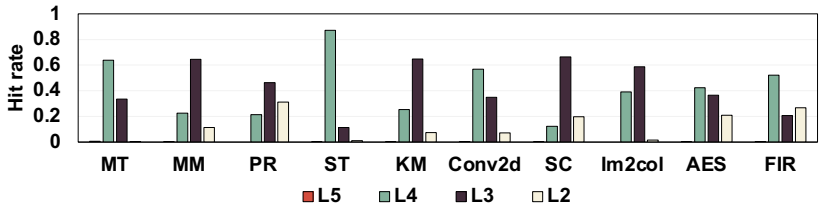

Figure 26: GMMU PW-cache hit rate in the baseline.

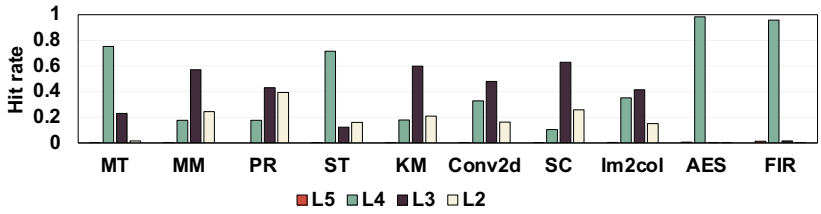

Figure 27: Host MMU PW-cache hit rate in the baseline.

the infinite PW-cache achieves up to 29.6% speedup, with an average performance improvement of 16.2%. To better understand the improvement, we report the hit rates for GMMU and host MMU PW-caches in Figure 26 and Figure 27, respectively. For GMMU PW-cache, 56.6% of page table walks can get the translation within 1 to 2 memory accesses and the rest of the translations must go through 3 to 4 memory accesses (i.e., walking the page table). For host MMU PW-cache, we observe 47.5% hit rates in the higher levels (e.g., L4) and 50.9% hit rates in the lower levels (e.g., L3 and L2). Therefore, multiple memory accesses are still required for address translation in the host MMU. Since the memory accesses are long latency operations, poor PW-cache hit rate can degrade the performance.

Implication of infinite PT-walk threads: With infinite page table walking threads in both GMMU and host MMU, all TLB misses are served immediately without waiting in the PW-queues. Referring to the second set of bars in Figure 25, we observe an average of 42.6% performance improvement over the baseline. The performance gain is higher for applications (e.g., `PR`, `SC`, and `MT`) whose waiting latency occupies higher percentages of the L2 TLB miss latency.

Implication of eliminating GPU page faults: Eliminating all GPU local page faults ensures that all translations are found in the GPU local page tables, leading to an average of 2.2× performance improvement (the fourth bar of each application in Figure 25). We quantify the *page-faults-per-kilo-instructions (PFPKI)* and show the PFPKI of each applica-

tion in Table 71. We observe that applications with substantial page sharing among multiple GPUs (detailed study in Section 4.1.3) have higher PFPKI values (e.g., `MT`, `PR`, and `SC`). This is because page sharing causes frequent page migration among GPUs, and hence subsequent page accesses to the migrated page will generate GPU local page faults. In contrast, applications with less page sharing across the GPUs (i.e., each GPU works on its own local data partitions) have lower PFPKI, such as `AES` and `FIR`. We also observe that applications with higher PFPKI achieve relatively high performance improvement (e.g., `MT` and `SC`), whereas lower PFPKI show less performance improvement (e.g., `AES` and `FIR`). However, `ST` has a high PFPKI but low performance improvement. This is because the contention in the host MMU page table walk is less severe in `ST` execution. In contrast, `Conv2d` has a low PFPKI but a high performance improvement. This is because a large number of pending requests are coalesced to the same page fault in L2 MSHR. Reducing the time for handling a page fault can significantly benefit the whole execution.

We show the performance improvement where we eliminate the data page migration latency while preserving the address translation latency (third bar in Figure 25). The results show an average of 63.9% performance improvement over baseline. Comparing the results with the above study, which eliminates the local page faults (2.2× performance improvement), one can observe that the address translation plays an important role in improving the performance, and there exists a large optimization potential.

4.1.3 Page Sharing Characterization

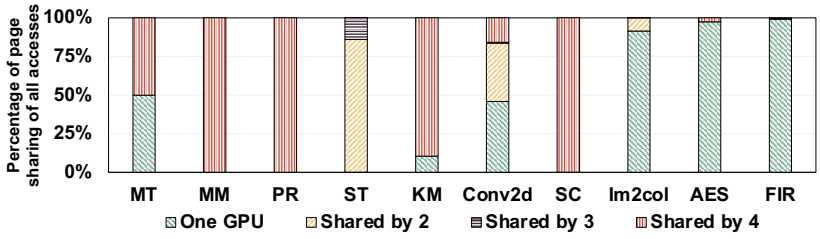

Figure 28: Percentage of page sharing.

Observation 1: *There exist substantial page sharing among multiple GPUs.* Figure 48 shows the page sharing among multiple GPUs during the execution of each application. Specifically, we define page sharing ratio as the percentage of shared page accesses divided

by the total number of page accesses during the execution of the application. As one can observe, a significant fraction of pages is shared by multiple GPUs. For example, in `MM`, `PR`, `KM` and `SC`, almost all pages are shared by all four GPUs. In `MT` and `Conv2d`, about half of the pages are shared between two or four GPUs. Such intensive page sharing also brings significant address translation sharing among the GPUs.

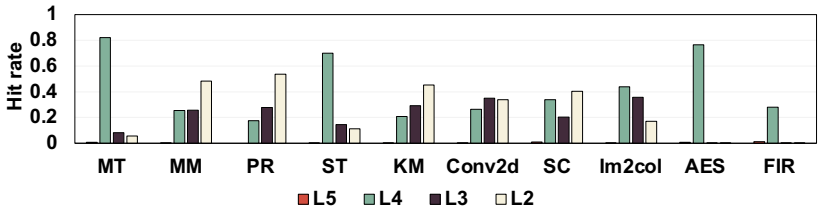

Figure 29: Remote PW-cache hit rate

Observation 2: *Local PW-cache misses and page faults can be resolved by other GPUs.* We investigate whether a local page fault can find the translation in other GPUs' PW-caches. The intuition behind is to leverage the translation reuses that stem from page sharing among GPUs to reduce the number of PW-cache misses and the latency of handling local page faults. To capture this opportunity, we define *remote* PW-cache hit as the percentage of local page faults that hits remote GPU's PW-cache. We observe 88.2% remote hit rate (all levels combined) in Figure 29. It is also important to note that, an average of 45.2% can hit in lower levels of PW-cache (e.g., L3 and L2), indicating that only 1-2 memory accesses are needed to get the translation from the remote GPU.

4.2 Trans-FW

4.2.1 High Level Overview

In this paper, we propose Trans-FW that leverages the substantial translation sharing among GPUs and employs the remote GPU to serve the translation requests when doing so is beneficial. There are three major challenges to implementing an effective and efficient remote forwarding scheme. First, the remote GPU supplies the translation request only if it holds the valid page. Therefore, it is important to determine which GPU has the valid page.

Second, accessing remote page tables may take longer than accessing the host page tables due to network congestion and remote PT-walk contention (e.g., PW-queue waiting and PW-cache misses). Thus, it is important to dynamically detect the beneficial scenarios and provide flexible and efficient hardware support for guiding and handling translation requests remotely. Finally, the proposed scheme should have minimal hardware overhead and is light-weight compared to enlarging the TLB capacity. To this end, we propose Trans-FW. Figure 30 shows the high-level overview of our design.

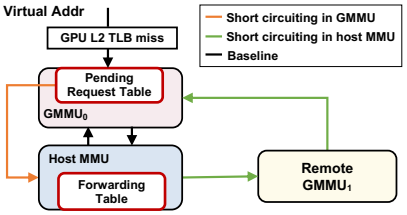

Figure 30: High level overview of Trans-FW.

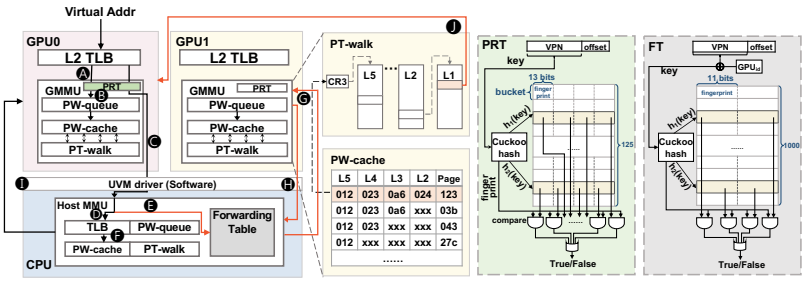

Figure 31: Translation lookup in Trans-FW, and details of PRT and FT.

4.2.2 Short Circuiting in GMMU

Recall our discussion in the GMMU PT-walk handling. For a given address translation request that misses the GPU L2 TLB, it will experience PW-queue queuing as well as PT-walk in the GMMU. The PT-walk latency can be large if the request misses the PW-caches. Suppose that the requested page is invalid in the GMMU page table, this request will be eventually a local page fault sent to the remote host MMU. Thus, the local PW-queuing and PT-walk add "unnecessary" latency to that request. To address this, we propose to "short circuit" the translation in GMMUs by eagerly sending the request to the host MMU upon an L2 TLB miss. Doing so avoids the aforementioned unnecessary latencies and can be particularly beneficial for requests that eventually cause local page faults. In other words, it allows the potential local page faults to be served early by the host MMU. However, the short circuiting approach may lead to an excessive number of requests being sent to the host MMU because the number of GPU L2 TLB misses is significantly larger than the GPU

local page faults. Therefore, a naive implementation of short-circuiting the GMMU may cause CPU-GPU interconnection congestion and PT-walk contention of the host MMU. To mitigate the contention, we propose a Pending Request Table (PRT) in each GMMU as a filter to reduce the number of requests sent to the host MMU. We next discuss the design of PRT.

Pending request table: The PRT tracks the translations of all pages that reside in the GPU's local memory. A Cuckoo filter [27] is implemented in the PRT, which is a hardware-efficient structure that supports fast set membership testing. Figure 53 shows the microarchitectural details of PRT (green box). Specifically, the PRT consists of 125 buckets and each bucket comprises four fingerprints. Each inserted item (i.e., VPN) is converted into a fingerprint using the cuckoo hash functions (i.e., h_1 and h_2, MetroHash hash function [33] is employed in our approach.). A fingerprint is formed by using the virtual page number. The Cuckoo filter provides efficient insertion and deletion operations. It inserts the fingerprint of an entry into one of the two alternative buckets indicated by the two hash functions. If neither bucket has space, the fingerprint selects one of the candidate buckets, kicks out the existing fingerprint, and re-inserts this victim fingerprint into its own alternate location. When looking up an item, the Cuckoo filter first calculates the item's fingerprint and the two candidate buckets. If any existing fingerprint in either bucket matches the request, the cuckoo filter returns true. Otherwise, it returns false. In our current design, the PRT comprises eight comparators so it can check all the fingerprint candidates from the two buckets in parallel. Deletion is supported by removing one copy of matched fingerprint from any candidate bucket. Note that, when two identical fingerprints are stored in the two checked buckets, a random one is selected and deleted, which causes false positive cases. Note that, the PRT needs to be updated when a page is migrated. Specifically, when a page is migrated away from the GPU, the virtual page number is used to locate the fingerprint. Then, this fingerprint is removed from the PRT. When a new page is migrated to the GPU, a new fingerprint is formed and inserted into the PRT. This update progress is off the execution critical path and can overlap with and be hidden by GPU execution.

Lookup procedure: When a translation request misses in the GPU local L2 TLB, the PRT is checked first. If the request misses the PRT, which indicates that the requested

translation is definitely invalid in the local GPU page table and the page is not presented locally (since the cuckoo filter has no false negative cases), then the request is early forwarded to the host MMU without a GMMU PT-walk. If the request hits the PRT, which indicates a high potential of finding a translation locally, then the request is sent to the GMMU PT-walk for local page table lookup. Note that, it can happen that the Cuckoo filter provides a false prediction and the GPU doesn't hold the translation. A local page fault will be generated after the GMMU PT-walk, which is the same as the baseline execution. However, considering a low false positive in our configuration, this scenario rarely happens and its additional latency has little impact on the overall performance.

4.2.3 Short Circuiting in Host MMU

Short circuiting the GMMU is beneficial only if the page fault requests can be handled timely in the host MMU. However, this is not always guaranteed. Requests in the host MMU might also experience long latency due to contention in the host MMU's PW-queue and PW-cache thrashing. Recall our observation in Section 4.1.3 where a significant fraction of translation prefix can be found in remote GPUs' PW-caches. To this end, we propose short circuiting the PT-walk in the host MMU by sending translation requests to remote GPUs if doing so is beneficial. Specifically, we propose to "borrow" the PT-walk in remote GPUs to avoid the contention and potential overheads in the host MMU. We introduce a structure, called Forwarding Table in the host MMU, and address the two important questions: *how to borrow?* and *when to borrow?*.

How to borrow: To borrow PT-walk from a remote GPU, it is important to first determine which GPU has the valid page. If a remote GPU does not hold the valid page, there is no performance gain since the remote GPU will also generate a page fault on the request. Therefore, we implement a Forwarding Table (FT) in the host MMU to indicate which GPU has the valid page. The FT leverages a similar Cuckoo filter design as in PRT to guide the forwarding of the requests to the remote GPUs. Figure 53 also shows the microarchitectural details of FT (grey box). Specifically, the FT has 1,000 buckets. The key used by the two cuckoo hash functions is a concatenation of VPN and GPU_{id}. Since each

bucket only has two fingerprints, we implement four comparators to perform parallel item lookups. A fingerprint is formed by concatenating the virtual page number and the owner GPU ID (i.e., the GPU who has the valid page) when the fingerprint is inserted into the FT. The FT is updated when a page is migrated. First, the page number and prior owner GPU ID are used to locate the fingerprint. Then, the old fingerprint is deleted from the FT and replaced with the new one, which is the concatenation of the page number and the new owner GPU ID. Note that, the FT supports four parallel GPU ID lookups. That is, four different GPU IDs are searched using one of two hash functions. If the desired fingerprint is found, it only takes one cycle. Otherwise, the four GPU IDs are searched using the other hash function. Note also that, identical fingerprints may store in the checked buckets, a random one is selected and deleted, the desired fingerprint may not be deleted. A new fingerprint is then inserted with the same page number but different owner GPU ID, which will result in two or more different owner GPU IDs for the same page number storing in the FT. As a result, when a request looks up the FT, it will return multiple owner GPU IDs. In the case, our design chooses any one and forwards the request to that GPU.

When to borrow: When a request arrives at the host MMU, the host MMU TLB and FT are searched in parallel. If the request hits in the host MMU TLB, the translation is directly returned to the requesting GPU. If the request misses in the host MMU TLB but hit in the FT, we will check the number of requests queued in the host MMU PW-queue and use it as the indicator for the host MMU PT-walk contention. Depending on the contention in host MMU PT-walk, two scenarios may happen. First, we observe when the number of queued requests is less than half of the PT-walk threads (we call it as the forwarding threshold)[1], a host MMU lookup is usually faster than a remote lookup, considering the remote GMMU contention and the network latency. In this scenario, we only rely on the host MMU to perform PT-walk without involving any remote GPUs. Second, if the number of queued requests is more than half of the number of PT-walk threads, the request is inserted into the host PW-queue but also forwarded to a remote GPU according to the FT in order to short circuit the PT-walk in the host MMU. When the desired translation is found in

[1]We also evaluate our approach with different forwarding thresholds (i.e., the number of queued requests) in Section 4.3.2

a remote GPU, the remote GPU will directly send the translation to the requesting GPU. The remote GPU will also notify the host MMU of success or failure after it performs the lookup. If the remote lookup succeeds, we check the host MMU PW-queue, if the request still waits in the PW-queue, the request is removed from the PW-queue to reduce the PT-walk contention. On the other hand, if the remote lookup fails due to the false positive in FT, the request will be discarded since the pending request has been sent to the host MMU PT-walk. Note that, it can happen that both host MMU and successful remote lookup will send the desired translation to the requesting GPU. However, the Trans-FW ensures that only the early returned translation will be used, and the latter one will be discarded. Note also that, in practice, we do not see many of these cases happen. This is because the queuing latency of the host PW-queue is generally large. In most cases, the host MMU will receive the forwarding GPU message and the request is removed from the host MMU PW-queue.

4.3 Evaluation

4.3.1 Overall Performance

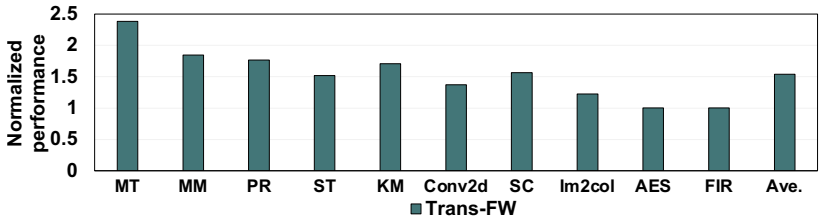

Figure 32: Normalized performance of Trans-FW.

Figure 32 shows the overall performance improvements of Trans-FW normalized to the baseline. Trans-FW achieves an average of 53.8% performance improvement across 10 applications. The performance improvements are significant for those applications with high PFPKI (shown in Table 71). For example, `MT` achieves over 2× over the baseline. In contrast, the performance improvements of `FIR` and `AES` are marginal. This is because these two applications are compute-intensive and the page fault latency can be hidden by the light-

weight context switching in GPUs. Also, these two applications have few page sharing and less local page faults. As such, these two applications are insensitive to page fault latencies.

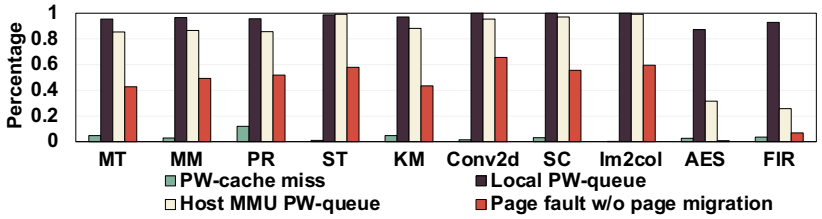

Figure 33: The reduced percentage of each latency component in Figure 24.

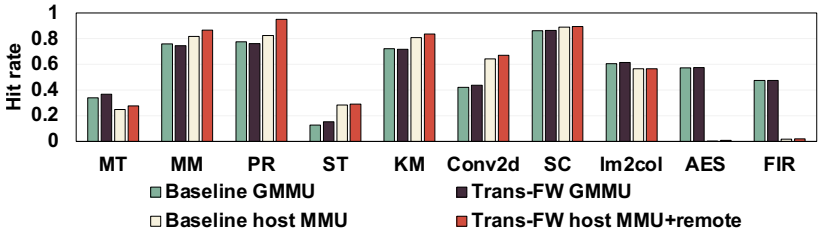

Figure 34: PW-cache hit rates at levels L2 and L3 in baseline and Trans-FW.

To understand the reasons behind the performance improvements, we plot the percentage reduction of each latency component quantified in Figure 24. One can make the following observations. First, Trans-FW significantly reduces the PW-queue waiting time by an average of 95.8% and 79.8% in GMMU and host MMU, respectively. The reductions in PW-queue waiting time directly translate to performance improvements. Second, Trans-FW reduces the latency of address translation parts of handling local page faults by 43.4%. Specifically, for 8 applications (except `AES` and `FIR`) that have substantial page sharing, our approach reduces the latency up to 53.3%. This is because page sharing among GPUs provides opportunities for local faults to be served by remote GPUs, thereby reducing page fault latency. Third, Trans-FW also reduces the latency caused by PW-cache misses. We also show average hit rates at L2 and L3 levels of both GMMU PW-cache and host MMU PW-cache in Figure 34. Note that, the host MMU PW-cache hits comprise the remote hit enabled by Trans-FW. As shown in Figure 34, compared to the baseline host MMU PW-cache L2 and L3 hit rates in Figure 27, Trans-FW achieves a higher hit rate. This is because the remote GPU has recently accessed the requested page, and a longer translation prefix is available in the remote PW-cache. However, the hit rate of GMMU PW-cache in Trans-FW is slighter lower than the baseline. This is because when serving remote requests, the newly inserted entry causes

the local PW-cache thrashing. Overall, Trans-FW improves the hit rates of PW-cache and achieves latency reduction caused by PW-cache misses.

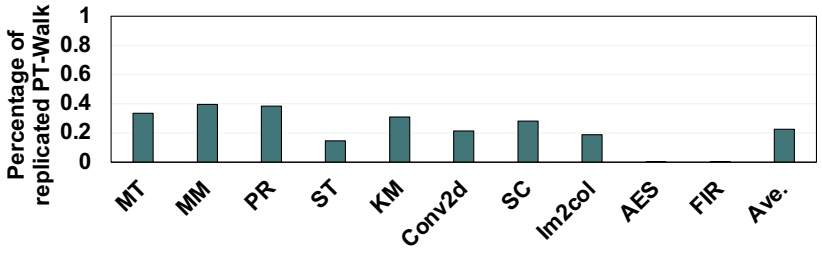

Figure 35: Percentage of replicated PT-walk to all host MMU PT-walk.

Recall that, when the number of queued requests in the host MMU PW-queue reaches a threshold, requests waiting in the host MMU PW-queue will also be forwarded to the remote GPU. This generates replicated PT-walk requests in the host MMU and the remote GMMU. Figure 35 shows the percentage of replicated PT-walk requests introduced by our approach to all host MMU PT-walk requests. One can observe that, our approach generates an average of 22.6% of replicated PT-walk requests to host MMU and remote GMMU. However, we want to emphasize that these additional PT-walk requests do *not* necessarily increase the overall total number of PT-walk memory accesses. The reasons are two-fold. First, short-circuiting in the host MMU allows the PT-walk requests to benefit the remote GPU's PW-cache. In our results, 65% of those replicated requests (i.e., 22.6%) in host MMU are eliminated before they incur any memory accesses because the translation is resolved by the remote GPU. Second, although those additional PT-walk requests cause 13.4% of additional PT-walk memory accesses in the GMMU (percentage calculated by the number of additional PT-walk memory accesses dividing the total GMMU PT-walk memory accesses), we short-circuit the local GMMU PT-walk for potential local page faults, which reduces an average of 49.6% of the total GMMU PT-walk memory accesses. Therefore, the total number of PT-walk memory accesses in GMMU is reduced by 36.2% (49.6% minus 13.4%) compared to the baseline. In summary, our approach reduces the total PT-walk memory accesses in both GMMU and host MMU.

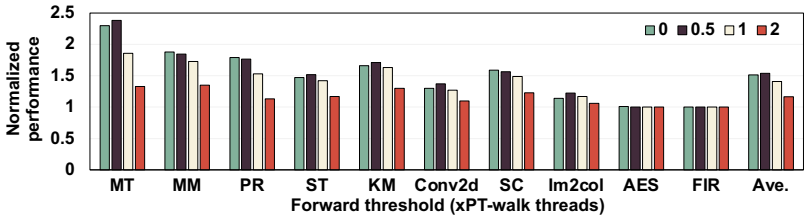

Figure 36: Remote forwarding threshold.

4.3.2 Sensitive Study

Forwarding threshold: Recall that, in our configuration of Trans-FW, the forwarding threshold is set to half of the PT-walk threads (Section 4.2.3). That is, when the number of queued requests exceeds half of the number of PT-walk threads, the request is forwarded to the remote GPU. In this study, we evaluate the performance impact under different forwarding thresholds in Figure 36. First, when the forwarding threshold is set to 0 (i.e., the request is forwarded right away when there are no immediate available PT-walk threads), the average performance improvement is 51.4% over the baseline, which is 2.4% lower than setting the forwarding threshold equal to 0.5. This is because it introduces more contention in the remote GPU PT-walk. Second, the performance improvement decreases as the threshold increases. The results show an average performance improvement of 41.0% and 16.7% when thresholds are set to 1 and 2. The main reason behind this is the increasing queuing time of host PW-queue.

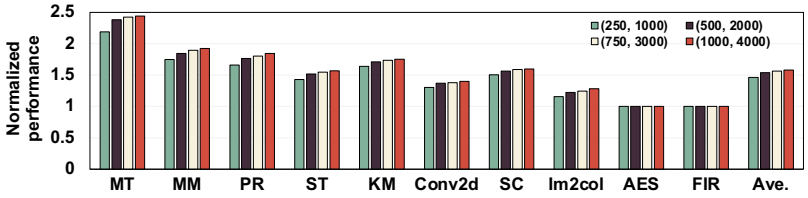

Figure 37: The performance of Trans-FW with different sizes of PRT and FT normalized to the baseline execution. The (x,y) on the legend indicates (size of PRT, size of FT).

FT and PRT size: We evaluate the performance of Trans-FW under different FT and PRT sizes in Figure 37. When PRT is 250 entries (i.e., fingerprints) and FT is 1000 entries (i.e., fingerprints), the average performance improvement is 46.3% on average. This is 7.5% lower than the default sizes (i.e., 500 in PRT and 2000 in FT). This is because more pages are mapping into the same fingerprints with smaller table sizes, which causes a higher false positive rate and results in more additional latency. When PRT and FT sizes are increased to

1000 and 4000 fingerprints, the average performance improvement is slightly higher than the default size. Considering the hardware overhead, we use 500 and 2000 as our configuration.

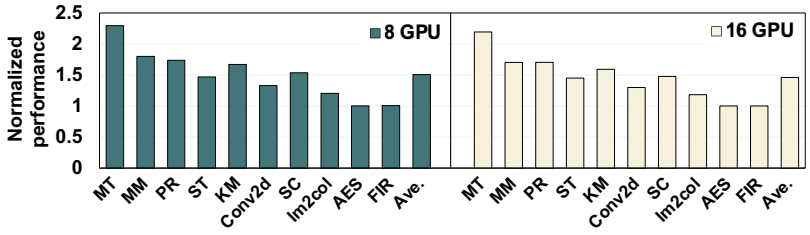

Figure 38: Performance of Trans-FW with 8 and 16 GPUs. Results are normalized to 8-GPU and 16-GPU baseline, respectively.

Number of GPUs: We evaluate Trans-FW in 8-GPU and 16-GPU systems to show its scalability. Note that, for a fair comparison, when we increase the number of GPUs, we do not increase the application input size. Figure 38 shows the performance of Trans-FW with 8 GPUs and 16 GPUs normalized to the baseline with 8 GPUs and 16 GPUs. We observe average improvements of 50.5% and 46.1% in 8 GPUs and 16 GPUs, respectively. The performance improvement is reduced as the number of GPUs increases. This is because, with more GPUs, more page faults will be generated and sent to host MMU due to the more frequent page sharing across GPUs, causing severer contention in the host MMU. Accordingly, the page faults handling time increases in the host MMU.

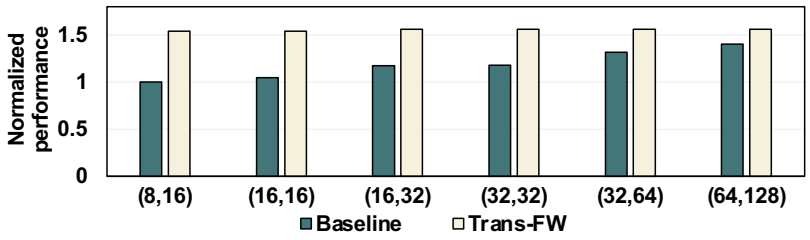

Figure 39: The performance of baseline and Trans-FW with different numbers of PT-walk threads normalized to the baseline execution. The (x,y) on the x-axis indicates (# of PT-walk threads in GMMU, # of PT-walk threads in host MMU).

Page table walk (PT-walk) threads: We evaluate Trans-FW under different numbers of PT-walk threads. Figure 39 shows the performance results. All the results are normalized to baseline execution with four-threaded PT-walk in GMMU and eight-threaded PT-walk in host MMU. One can make the following observations. First, Trans-FW outperforms

the baseline in all configurations, achieving an average performance improvement of 56.8%. Second, increasing the number of PT-walk threads improves the performances in both the baseline and Trans-FW. This is because the PW-queue waiting time is effectively reduced, and the baseline benefits more from the increasing PT-walk. Finally, Trans-FW with fewer PT-walk threads achieves higher performance than baseline with more PT-walk threads. For example, Trans-FW (second bar) in (8,16) configuration outperforms the baseline (first bar) in (64,128) configuration by 13.4%. This is because Trans-FW not only reduces the PW-queuing latency, but also reduces the PW-cache miss penalty and the unnecessary PT-walk latency caused by local page faults. This also indicates that simply employing a large number of PT-walk threads in the baseline cannot achieve comparable performance with our approach.

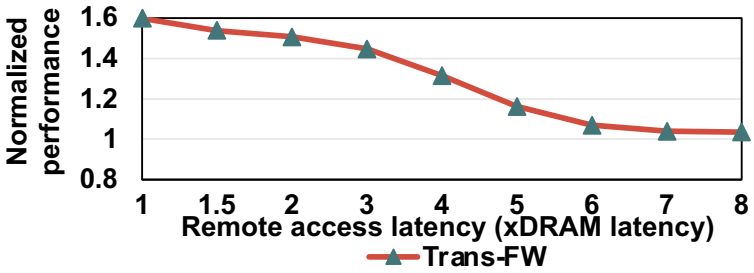

Figure 40: Remote access latency.

Remote access latency: We evaluate Trans-FW under different remote GPU access latencies to show the cross-over point when accessing a remote GPU compared against invoking the PT-walk in the host MMU. Figure 40 shows the performance of Trans-FW when varying the GPU interconnection latency to multiple times of GPU memory latency normalized to baseline execution. One can observe that, the performance improvement of Trans-FW decreases when the remote access latency increases. In particular, when the remote latency is beyond 8× of the GPU local memory latency, the performance of accessing the remote GPU is almost the same as accessing the page table in the host MMU. This is because, at this point, the remote access latency is comparable to the waiting latency of PT-walk. Note that, the remote latency is typically 100-150 cycles as employed by prior works [30, 28, 1], we use 150 cycles in the main evaluation.

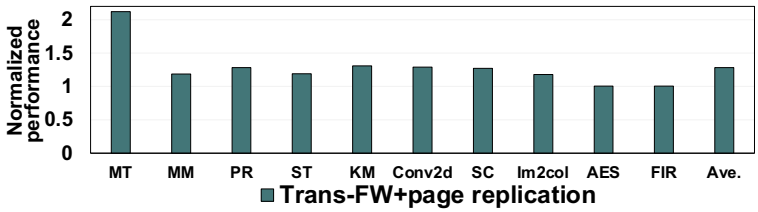

Figure 41: Performance of Trans-FW with page replication normalized to baseline with page replication.

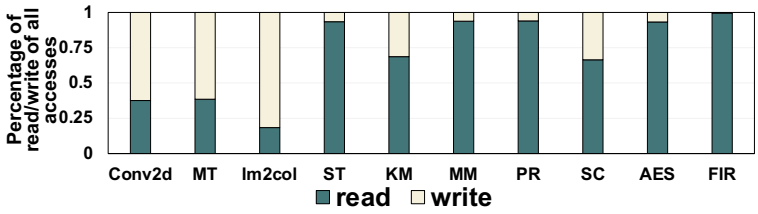

Figure 42: Percentage of reads and writes to all shared pages.

4.3.3 Page Replication

The unified virtual memory in modern GPU systems supports read-replication, in which a page is allowed to be accessed and replicated in multiple GPUs' physical memory with different physical addresses [52]. When a local page fault happens, a read-only page is replicated in the faulting GPU physical memory. To ensure consistency, when a GPU performs a write operation, it invalidates all the page replications in other GPUs, similar to cache coherence protocol. An ESI (Exclusive, Shared, and Invalid) memory coherency protocol is employed. When a write to a read-replicated page (S state) occurs, a protection fault is triggered. The GMMU dispatches an interrupt to the CPU to invoke the fault handler. The handler then invalidates the page from all other owners and shoots down all stale TLB entries in both the host MMU and GMMU except for the GPU that executes the write operation. Once the GPU receives the completion message of fault handling, it will re-execute the write. We evaluate Trans-FW under read-replication. Figure 41 illustrates the performance of our approach with page replication normalized to baseline execution with page replication. We observe an average of 28.4% performance improvement brought by Trans-FW. Comparing the results with the performance in Figure 32, the improvement is less. This is because that read replication effectively reduces the number of local page faults and potentially improves the PW-cache hit rates. However, for some applications (e.g., `MT`, `Conv2d`, and `Im2col`),

our approach still significantly outperforms the baseline read-replication. To understand the reason behind, we quantify the read operations and writing operations to shared pages across GPUs in Figure 42. We observe that read-replication has marginal improvements for write-intensive applications due to the frequent write-invalidation (e.g., `MT`, `Conv2d`, and `Im2col`). In contrast, our approach is able to improve applications exhibiting both write-intensive and read-intensive page sharing.

4.3.4 Remote Mapping

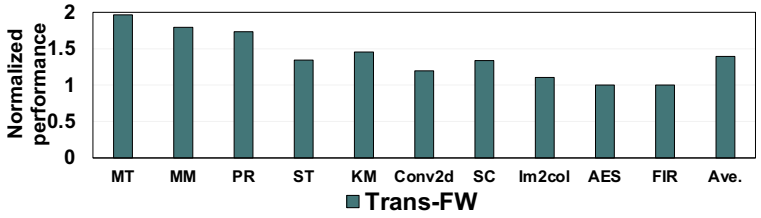

Figure 43: Performance of Trans-FW with remote mapping normalized to baseline with remote mapping.

Our discussion so far leverages the on-touch migration which is the default scheme in modern GPUs [52, 48]. UVM also supports another page migration scheme: remote mapping. That is, a GPU can establish a direct mapping and access the remote GPU's physical memory without actually migrating the page. Instead of migrating a page, the first access to a page that is not locally available will generate a page fault and then create a corresponding page table entry that points to the page in remote GPU memory. Pages are only migrated after reaching accessing thresholds. When a page migration happens, the translations that map to that page from different GPUs have to be invalidated to ensure translation coherence across GPUs. We evaluate Trans-FW with remote mapping. Specifically, we leverage the access counter as in recent NVIDIA GPUs [52, 50] to determine the page migration. Figure 43 shows the performance of our approach with remote mapping normalized to baseline execution with remote mapping. The results indicate that Trans-FW achieves an average of 39.3% performance improvement. This demonstrates Trans-FW works with remote mapping and improves the performance when using remote mapping. The improvement is less compared to the results in Figure 32. The main reason is that remote mapping helps reduce the

number of local page faults and the amount of page thrashing, especially for applications with substantial sharing (e.g., KM, SC). As a result, the host MMU PT-walk contention is reduced.

4.3.5 Adopting Large-sized Pages

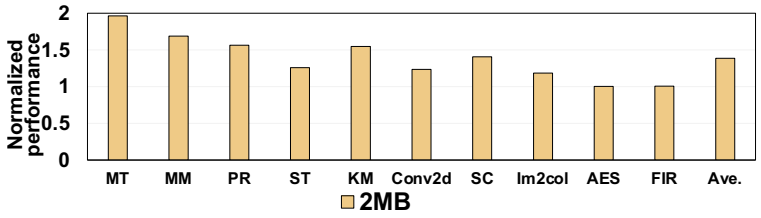

Figure 44: Performance of Trans-FW with 2MB page normalized to baseline with 2MB page.

We next evaluate Trans-FW with 2MB page size. Figure 44 shows the performance of Trans-FW normalized to the baseline execution with 2MB page size. Trans-FW achieves an average of 38.6% performance improvement. This is less compared to the 4KB page size results. The reason is that, in the baseline execution, choosing a large page size effectively increases the L2 TLB hit rate, thus mitigating the contention in GMMU PT-walk. However, it is important to note that Trans-FW still achieves substantial performance improvements. This is because, by short-circuiting the PT-walk in GMMU and the PT-walk in the host MMU, Trans-FW is able to further reduce the PT-walk latency. Besides, although adopting large-sized pages reduces local page faults, it increases false sharing and causes extra inter-GPU page faults when a large page is frequently shared among different GPUs.

4.3.6 Comparison to PW-cache Prefetching

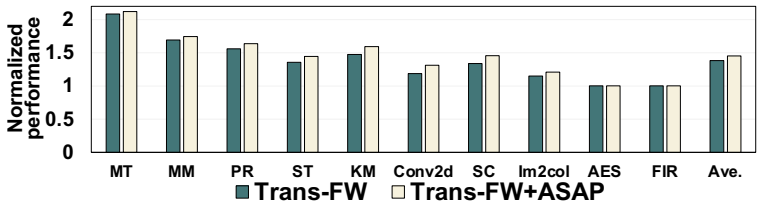

Figure 45: Comparison to ASAP [44] PW-cache prefetching.

PW-cache prefetching is another well-known technique to improve the address translation efficiency by reducing the number of page walks. We compare Trans-FW with the state-

of-the-art *ASAP* [44] address translation prefetching. The ASAP prefetching focuses on prefetching lower levels (i.e., L2) of page table during page table walk. In our comparison experiments, we employ ASAP in both GMMU PT-walk and host MMU PT-walk. Figure 45 shows the performance of Trans-FW and Trans-FW+ASAP normalized to the ASAP. We make the following observations. First, Trans-FW outperforms ASAP by an average of 38.4%. The reason is that ASAP does not mitigate PT-walk contention nor page fault handling latency. In contrast, our Trans-FW short circuits PT-walk caused by local page faults, mitigates the contention in PT-walk, and reduces the latency of handling page faults. Second, our approach can be combined with PW-cache prefetching to further improve the address translation efficiency. Specifically, Trans-FW+ASAP achieves 45.2% improvement over the ASAP. This is because the PW-cache miss latency is further reduced when combined Trans-FW and ASAP. In summary, the results demonstrate that the proposed Trans-FW is flexible to work with PW-cache prefetching strategies.

5.0 IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations

5.1 Motivation

Migrating data from remote memory to local memory enables data to always be accessed at local memory access bandwidth rather than remote memory access bandwidth (which is limited by interconnect bandwidth). Thus, page migration is crucial to address Non-Uniform Memory Access (NUMA) bottlenecks in multi-GPU systems.

Page migration modifies the existing virtual to physical mapping. Consequently, all system-wide data structures (i.e., per-GPU TLBs and per-GPU local page tables) containing the old virtual-to-physical mapping must be invalidated. To perform this task, conventional UVM drivers simply broadcast page table invalidation requests to all GPUs in the system (even those GPUs that never requested the translation). In response, GPUs receiving the page table invalidation requests walk their respective local page table and invalidate the corresponding PTE (even if it were invalid to begin with).

Broadcasting page table invalidation requests works well when page migrations are infrequent. However, when applications exhibit high page sharing between multiple GPUs (quantitatively analyzed in Section 5.1.2), frequent page migrations significantly increase the number of page table invalidation requests to invalidate GPU local PTEs. These invalidations contend with existing demand TLB misses thereby increasing their miss latency. Demand TLB miss latency is performance critical and must be reduced to improve GPU performance.

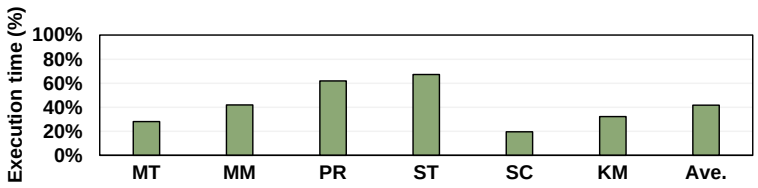

Figure 46: Page table invalidation overhead.

Figure 46 demonstrates the page table invalidation overheads for representative multi-

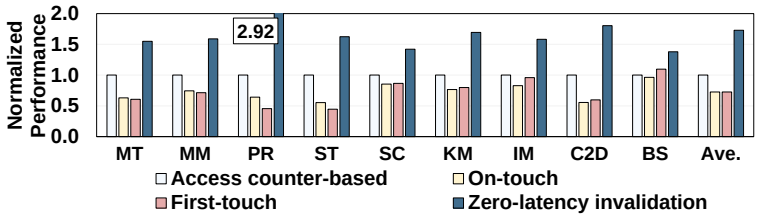

Figure 47: Performance of each scheme relative to access counter-based migration.

GPU applications[1] running on a 2-GPU NVIDIA A100 system.We use *uvm-eval* [77] to profile the GPU page table invalidation time. The figure shows that nearly half (average 42%) of total execution time is spent on handling page table invalidations. Applications with high page sharing (e.g., `PR` and `ST`) tend to have higher page invalidation overheads. These results are consistent with previous GPU UVM studies [4, 21] and strongly motivate the need for substantially reducing the page invalidation overheads.

To illustrate the performance opportunity from eliminating page invalidation overheads altogether, Figure 47 presents a simulation-based study that depicts the performance of an ideal page migration policy that incurs no page table invalidation overheads. That is, the per-GPU page table invalidation requests incur *zero* latency and bandwidth contention with the PTEs being updated instantaneously. We use an industry-validated multi-GPU simulation framework (MGPUsim [72]. Our simulated baseline system employs *access counter-based page migration* (which is the baseline on NVIDIA A100 GPUs [53]), where pages are migrated only when remote accesses reach a given threshold. The figure shows that an idealized system with zero page table invalidation overheads outperforms the baseline system by 38%-1.92× (average 73%). For reference, we also report the performance of (a) *first-touch migration* where the page is pinned to the GPU that first accessed a page with no further migrations (b) *on-touch migration* where pages are always migrated to the requesting GPU. While both these page migration policies can have lower page table invalidation overheads, we observe that they generally perform worse than the access counter-based migration policy either because of the increased number of remote memory accesses or frequent page migrations.

The results from Figure 46 and Figure 47 clearly demonstrate the importance of reducing page table invalidation overheads. Such overheads stem from page table walk contention

[1]These are multi-GPU ready applications with large input sets running on real hardware and are compatible with *uvm-eval* [77].

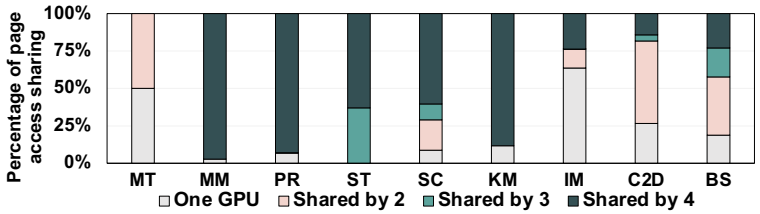

Figure 48: Distribution of accesses referencing shared pages.

between demand TLB miss requests and page table invalidation requests. Reducing page invalidation overheads can be accomplished by (i) *reducing the number of invalidations sent* by employing a directory to only send page table invalidation requests to GPUs that hold the corresponding address translation (ii) *reducing the invalidation latency* by batching invalidation requests to exploit spatial locality between multiple invalidation requests. Before we discuss our solution, we provide a detailed discussion of address translations in multi-GPU systems and quantitatively investigate the contentions caused by page table invalidations (Section 5.1.1).

5.1.1 Page Sharing and Overheads

We now show that substantial page sharing is the root cause for invalidations and provide a detailed contention analysis.

5.1.2 Multi-GPU Page Sharing Characterization

In a multi-GPU system, PTE invalidations are triggered during page migration when a page is frequently referenced by multiple GPUs. As such, we first investigate the page sharing behavior of applications in a multi-GPU environment. Page sharing in multi-GPU applications is common as threads running on different GPUs may access the same data structures. In this paper, we define the page access sharing ratio as the ratio of total shared page accesses to the total accesses. Figure 48 shows that there exists significant page sharing among multiple GPUs. For example, in `MM`, `PR` and `KM`, almost all accesses are to pages shared by all GPUs. In `MT`, `C2D`, `BS`, a large fraction of accesses is concentrated on the pages that are shared by 2 GPUs. This is because, `PR` has random access pattern where any GPU needs to both read and write data from/to the entire GPU address space. `C2D` needs to access input

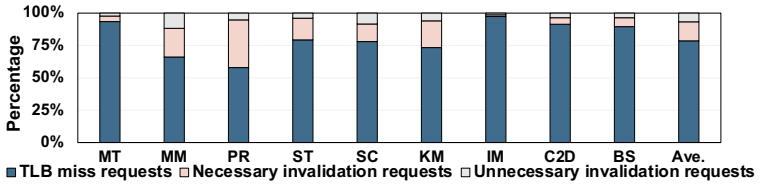

Figure 49: Percentage of invalidation requests and demand TLB miss requests.

data from surrounding indices that are resident on other GPUs.

5.1.3 Page Table Walk Characterization

Recall that when a page needs to be migrated in the remote mapping approach, all GPUs need to perform local page table walks to invalidate their corresponding page table entries before the page can be migrated. Figure 49 provides the distribution of requests to the page walker in terms of PTE invalidation requests (necessary and unnecessary) and demand TLB miss requests. One can observe the number of invalidation requests accounts for a quarter (i.e., 27.2%) of total requests to the page walker. For applications with excessive page access sharing, they have a higher percentage of invalidation requests, such as `MM`, `PR`, and `KM`. This is because a large number of pages are heavily accessed by multiple GPUs, which causes a large number of page migrations and thus significant PTE invalidation requests.

We also show the percentage of unnecessary PTE invalidation requests in Figure 49. Specifically, since each migration broadcasts invalidation requests to all GPUs, those GPUs that have not accessed the page (or their corresponding PTEs are already invalid) still need to perform page table walks. This is because they do not have information on whether the local PTE is valid without actually walking the page table. We refer to these invalidations as unnecessary invalidation requests. On average, the results in the figure indicate that nearly one-third (i.e., 32%) of PTE invalidations broadcasted are unnecessary.

We now study the performance impact caused by PTE invalidation request contention. We identify two latencies that are affected by PTE invalidation requests: (i) demand TLB miss request latency, defined as the address translation latency of requests that miss L2 TLB, and (ii) page migration waiting latency, defined as the latency between a page receiving a migration request and the actual migration of the page.

Figure 50 assumes a hypothetical system where the PTE invalidation requests incur no

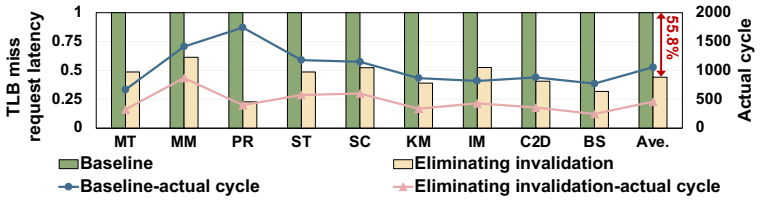

Figure 50: Demand TLB miss requests latency in baseline and zero-latency invalidation execution (normalized values and the averaged exact number of cycles).

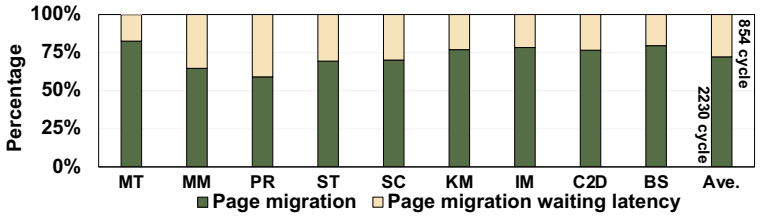

Figure 51: Percentage of page migration latency and page migration waiting latency.

contention to demand TLB requests. In such a system, when the GPU receives invalidation requests from the UVM driver, all latency and contention from invalidation requests are removed, including waiting in the page walk queue, page walk cache lookup, and performing page table walks. The figure shows the demand TLB miss request latency of such a system normalized to the demand TLB miss request latency in the baseline system with PTE invalidations. We also plot the actual values of averaged cycles shown as lines in the figure corresponding to the right y-axis. The result shows that without the interference of invalidation requests, the demand TLB miss request latency is on average reduced by 55.8% of the baseline. This is because invalidation requests follow the same page table walk process as demand TLB miss requests, which includes waiting in the page walk queue, lookup the page walk cache, and walking the page table after the page walk cache miss. Therefore, the invalidation requests (i) extend the page table walk queuing time for demand TLB miss requests, (ii) thrash the page walk cache, which may reduce the page walk cache hit rate for demand TLB miss requests, and (iii) increase the contention of page table walk threads in the GMMU.

Once a page is determined to migrate, those requests accessing that page will need to wait for the page migration to complete and establish new translation mapping before those requests can access the page. Since the invalidation delays the page migration process

(specifically, translation resolve time), it also extends the waiting latency of those requests. Figure 51 plots these extra page migration waiting latency caused by invalidation requests. We observe the extra waiting latency is 38.3% of the page migration latency. However, we want to mention that, not all these waiting latencies contribute to performance degradation because they are not on the critical path and can be hidden by the computation context switching in GPU. But also note that, for memory-intensive applications, there is not much computation to hide these latencies and they play a sizable role in the overall performance.

5.2 In-PTE Directory and Lazy Invalidation (IDYLL)

5.2.1 High Level Overview

The primary goal of our work is to retain the benefits of state-of-the-art counter-based page migration policy while addressing the overheads associated with frequent page table invalidation requests. We propose IDYLL that consists of two mechanisms: (i) a software in-PTE directory invalidation mechanism to reduce unnecessary page table invalidations, and (ii) a hardware mechanism called Invalidation Request Merging Buffer (IRMB), which implements an invalidation batching scheme to amortize invalidation overheads and a lazy update of page table to minimize the contention between demand TLB miss requests and invalidations. However, there are three challenges to implementing IDYLL. First, the UVM driver should only send the invalidation requests to those GPUs that have valid translation mappings instead of broadcasting. Therefore, it is important to record which GPUs have valid mappings. Second, it is crucial not to perform invalidations blindly upon receiving invalidation requests, so as to reduce unnecessary contentions with the demand TLB miss requests and page migration wait time. Finally, the proposed IDYLL should involve minimal hardware overheads.

5.2.2 In-PTE Directory Invalidation

Section 5.1.3 showed that a large fraction of PTE invalidations are unnecessary. We propose to filter the needless PTE invalidation requests using an In-PTE Directory Invalidation design on the host side. This approach avoids contention and queuing delays in the page walk queues, page walk caches and further reduces several operations on those GPUs that do not hold a valid translation to begin with. We now discuss our In-PTE Directory Invalidation design in detail.

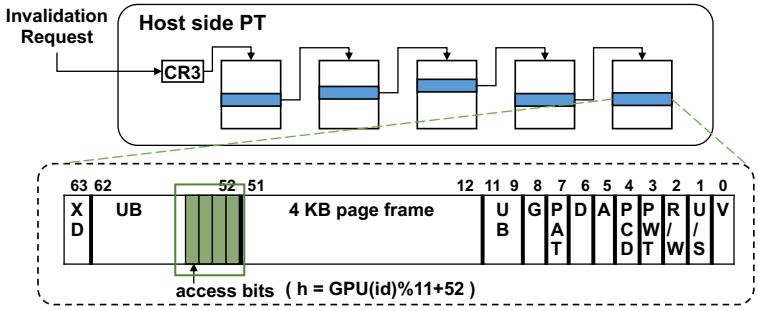

Figure 52: Page table entry format for 4 KB pages.

Which GPUs to invalidate? The In-PTE Directory Invalidation should be aware of all translation mappings in each GPU and which GPU has these valid translation mappings. However, keeping track of such a high volume of entries can cause a lot of memory overhead. Fortunately, the host-side page table holds all valid and up-to-date address translations for all GPUs. We only need to add information of which GPUs hold these valid mappings to the corresponding address translation entries. To this end, we leverage unused bits in the host-side PTEs as *access bits* for each GPU to store the GPU access information. Figure 52 shows the page table entry format in 4 KB pages[2]. Specifically, bits 51 to 12 store the physical page number for the virtual address, bits 62-52 and 11-9 are unused bits, and the remaining bits store metadata about the page. Note that, the maximum number of unused bits in the current PTE format is 14 bits. When the number of GPUs exceeds the maximum unused bits, the GPUs access bits cannot be mapped one-to-one with the unused bits. Therefore, we use a modular hash function to map multiple GPUs access bits to one unused bit. To simplify the calculation, we only use unused bits 62-52 for access bits. Specifically, the hash

[2]Unused bits stay constant with different page sizes [16].

function is h(GPU_{id}) = GPU_{id} % m + 52, where m represents the number of unused bits used for access bits, which is equal to 11 in our design[3]. For example, in our default 4-GPU system, the unused bits 55-52 of PTE correspond to the access bit of GPU_3-GPU_0 (as shown in the green box of Figure 52). Mapping multiple GPU access bits into a single slot can lead to false positives, which only lead to unnecessary requests being sent to the GPU, but do not affect correctness. Nevertheless, the number of unnecessary requests sent to the GPU is significantly reduced compared to the baseline. Initially, all access bits in the host-side PTE are set to 0. When a GPU, for example, GPU_0, accesses a page for the first time, it will miss all local GPU TLBs and the local page table entry of this page is also invalid. A far fault is generated and sent to the host side. The host-side page table is then walked to get the desired address translation. At this point, the access bit for GPU_0 in the host-side PTE (the unused bit 52) is set to 1, as GPU_0 will establish a valid mapping to its page table when the address translation is replayed.

Lookup procedure: The UVM driver, upon receiving a page migration request, performs a page table walk in the host-side page table to invalidate the corresponding mapping, as well as obtain the access bits information. Then, the driver sends the invalidation request only to GPUs with access bits set to 1. The access bits are also cleared to 0, as the corresponding remote mappings in each GPU will be invalidated to ensure translation coherence. Note that in the baseline, the invalidations are broadcasted before the host-side page table walk is completed. In our design, we must wait since we leverage the host-side page table walk to determine which GPUs should be sent the invalidation requests, thus adding additional latency in sending invalidation requests. However, we emphasize that even in the baseline, the host side has to perform a page table walk to invalidate its corresponding PTE. Sending the invalidation requests early does not bring significant performance and this additional latency is marginal to the overall performance. Nevertheless, we include these overheads in our later evaluation.

After the page migrates to a new GPU, a new translation mapping is established. The new mapping also needs to be updated in the host-side page table. The corresponding access bit is set to 1 when the host PTE is updated.

[3]We evaluate scalability with 4 unused bits in Section 5.3.2.

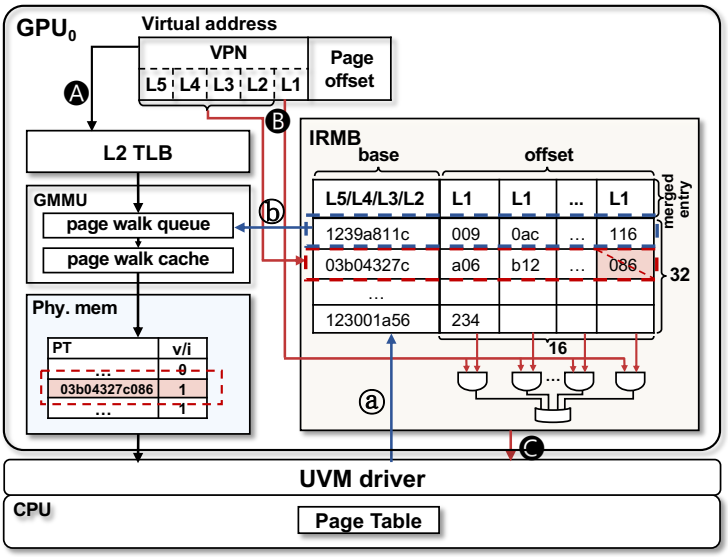

Figure 53: Overview of IDYLL.

5.2.3 Lazy Invalidation

In-PTE Directory Invalidation eliminates unnecessary invalidations while helping little with those GPUs that hold the valid translations and must perform the page table invalidations. Due to the significant page sharing in multi-GPU, there are still substantial valid translation mappings that need to be invalidated when performing page migrations. These invalidation requests introduce extra latency for both existing demand TLB miss requests and page migration waiting time as specified in Section 5.1.3. Therefore, we design a lightweight hardware component called *Invalidation Request Merging Buffer* (IRMB) in each GPU. The IRMB acts as a "valve" for the invalidation requests by temporally buffering incoming invalidation requests and lazily updating the local page table with minimum impact to existing demand TLB miss requests. Note that, we only perform lazy updates to the PTE while keeping the TLB shootdown process as it is in the baseline. That is, upon receiving an invalidation request, the TLB is immediately invalidated.

Invalidation Request Merging Buffer (IRMB): The IRMB tracks the virtual page number (VPN) of invalidation requests that are received from the UVM driver. In particular, we observe that in many applications, pages being migrated are nearby to each other in the address space. Therefore, the virtual addresses (VAs) of invalidation requests are also nearby and share a significant number of identical bits in the addresses. Based on this

observation, we design compressed entries for IRMB. The key idea behind this is to exploit the similarity of VAs where the VPN of invalidation requests in the IRMB at a certain period has a large number of identical bits. These identical bits can be merged so that IRMB can accommodate more invalidation requests. Besides, the nearby invalidations that are merged into one entry can leverage the same page walk cache when updating the invalidations to the page table to amortize the invalidation overheads. Therefore, in our design, we merge the invalidation request with the same high-level VPN into one entry. Figure 53 illustrates the microarchitecture of the IRMB. Specifically, the VPN of a page is partitioned into a 36-bit base (L5-L2 level of VA), and a 9-bit offset (L1 level of VA). The VPN with the same base coalesces into one *merged entry*. The IRMB consists of 32 merged entries with different bases, and each merged entry comprises 16 different offsets[4].

IRMB insertion and eviction: When a GPU receives the invalidation request from the UVM driver, bits L5-L2 of the VA of the invalidation request are used to match the bases. If the bits match the base in IRMB, the L1 bits of the VA are inserted into the merged entry. Otherwise, a new merged entry is created with the bits L5-L2 of the VA of the invalidation request as the base, and the L1 bits are inserted into the new-created merged entry (ⓐ). It can happen that the IRMB is full during invalidation request insertion, and the eviction of the merged entry is required. First, if the base is full, we use the LRU replacement policy which evicts the least recently used merged entry (ⓑ) and uses the slot to store the newly-arrived invalidation request. The reason for choosing an LRU merged entry is that, if a page is recently migrated, there is a high probability that its neighboring pages will be migrated later due to the data access locality. Therefore, if a PTE needs to be invalidated, its neighboring PTE may also be invalidated later, so we can keep this merged entry in the IRMB to coalesce more invalidation requests and invalidate them in once. Second, if the offset is full, we evict all offsets in the corresponding merged entry and insert the L1 bits into this entry. Note that, all the eviction will trigger the invalidation to propagate to the page table.

IRMB writeback: In our design, we use the IRMB to temporarily record the invalidation request first. It is crucial to propagate invalidations without introducing significant

[4]We also evaluate different IRMB sizes in Section 5.3.2.

latencies to the critical path execution. Ideally, the invalidations should overlap with normal executions so that the invalidation overheads could be hidden. Therefore, first, in our design, when the page table walker is available, we invalidate the LRU merged entry corresponding PTEs and also evict this merged entry in the IRMB. In such a case, the invalidation will neither affect demand TLB miss requests nor page migration. Second, in the case of the IRMB being full, the propagation of invalidations is essential. Therefore, when an eviction happens in the IRMB, all PTEs corresponding to VAs in the evicted merged entry are invalidated sequentially. This allows the invalidations to be handled in a batch, and improves the L2 level page walk cache hit rate since all these invalidations have the same higher level of the VA.

IRMB lookup: Figure 53 also illustrates the lookup procedure in the IRMB. Specifically, When a translation request misses GPU L1 TLB, the L2 TLB (Ⓐ) and the IRMB (Ⓑ) are searched in *parallel*. Three different scenarios may happen. First, if the request hits the L2 TLB, the lookup in IRMB is abandoned, and the request is handled the same as the baseline. Second, if the request misses the L2 TLB and also misses the IRMB, which implies the desired PTE is up-to-date, whether valid or invalid. Therefore, the request behaves the same as the baseline, waiting in the page walk queue, looking up the page walk cache, and walking the page table. Third, if the request misses the L2 TLB but hits the IRMB, which indicates that the translation mapping in the page table cannot be used as it should be invalidated. Then the request bypasses the local page table walk and directly raises a far fault to alert the UVM driver (Ⓒ). By doing so, the latency of demand TLB miss requests that eventually cause far faults is further reduced. Note that the corresponding PTE invalidation process may also be bypassed in this case. This is because a far fault is generated, and a new translation mapping will be received after the host-side page table walk. Therefore, we can update the PTE directly after a new mapping is replayed without invalidating it. It can happen that the corresponding entry is evicted from the IRMB before the new mapping is received due to capacity conflicts. In this case, the evicted entries follow the IRMB eviction process and perform the PTE invalidation in the page table, even though these invalidations are unnecessary. Note that, before a new mapping is received, there won't be any subsequent requests to the same page being sent to GMMU for page table walk. This is because the

original request that triggers the new mapping resides in the L2 TLB MSHR. All subsequent requests to the same page will be blocked and held at the L2 TLB MSHR. Therefore, if the corresponding entry is not found in the IRMB and a new mapping is not received, it is guaranteed that the request does not access a stale translation. Note also that, upon receiving a new mapping, the IRMB is checked if the corresponding VPN is present in the IRMB. If it is not in the IRMB, the new mapping is directly inserted into the page table walk queue for PTE update. If it is found in the IRMB, the particular offset from the merged entry in IRMB is removed, since this translation mapping is established in the page table as a valid translation.

5.3 Evaluation

5.3.1 Overall Performance

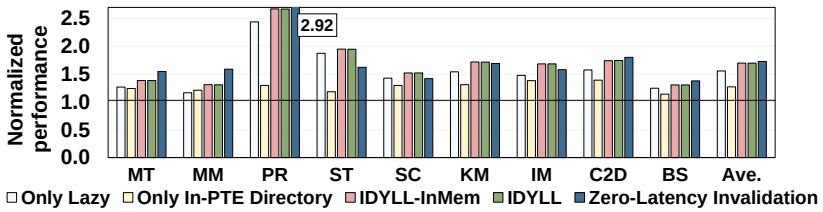

Figure 54: Performance of each scheme relative to baseline.

Figure 54 shows the performance improvements of using Lazy Invalidation only, In-PTE Directory Invalidation only, and our proposed IDYLL normalized to the baseline. We use the end-to-end execution time of the application to compute the normalized performance. One can make the following observations. First, only using the In-PTE Directory Invalidation achieves an average of 27.3% performance improvement over the baseline, and only using Lazy Invalidation achieves an average of 55.8%. With both optimizations enabled, IDYLL can provide an average improvement of 69.9%. The improvement brought by IDYLL is smaller compared to the sum of the improvements achieved by each optimization individually. This is because the two optimizations are complementary to each other and both reduce the contention with demand TLB miss requests. Second, the performance improve-

ments of the two mechanisms on different applications vary. Many applications achieve higher performance improvement with Lazy Invalidation, as it reduces the latency and contention caused by all invalidation requests. In contrast, In-PTE Directory Invalidation only works for unnecessary invalidations. However, one can observe that MM benefits more from In-PTE Directory Invalidation. This is because the unnecessary invalidation requests thrash the IRMB in Lazy Invalidation and removing these unnecessary invalidations is immensely helpful. Finally, with both mechanisms enabled, the performance improvement is significant for high MPKI applications (shown in Table 71). For example, PR achieves 2.67× over the baseline. In contrast, the performance improvement of BS is moderate. However, for MT, which has the highest MPKI value while not achieving an expected performance improvement. This is because the percentage of invalidation requests in MT is much lower as shown in Figure 49. As such, invalidation requests have less impact on its performance. We also find that IM has a relatively lower MPKI value and a small percentage of invalidation requests, but achieves high performance improvement. This is because IM has a memory-intensive process of converting each patch of image data into a column, where handling page table walk latency can hardly be hidden by the computation context switching (e.g., warp scheduling) in GPUs. Reducing invalidation latency can significantly benefit the execution.

Figure 54 also shows the performance improvement of zero-latency invalidation normalized to baseline (the last bar). Our approach achieves a comparable performance improvement against zero-latency invalidation. Interestingly, we find that the performance improvements for some applications in our approach (e.g., ST, SC, and IM) are higher than the performance improvements of zero-latency invalidation. The reasons are twofold. First, our In-PTE Directory Invalidation reduces unnecessary invalidation requests being sent to GPUs, whereas in zero-latency invalidation, all invalidation requests are still sent to all GPUs. Therefore, our approach reduces interconnect congestion. Second, the IRMB in our design also serves as an indicator of invalid PTE. Thus, when a demand TLB miss request hits in the IRMB, we can bypass the local page table walk and directly send the request to the host (since the requested PTE would have become invalid in the local page table had we not delayed the invalidation). This reduces the page table walk required by the demand TLB miss compared to the zero-latency implementation, thereby reducing overall demand

TLB miss request latency.

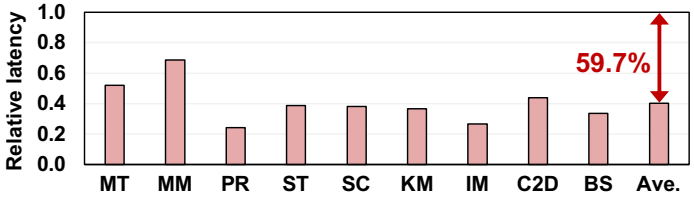

Figure 55: Demand TLB miss request latency.

Demand TLB miss request latency: To understand the reasons behind the performance improvements, we plot the demand TLB miss latency and page migration waiting latency when using IDYLL. Figure 55 shows the total latency of demand TLB miss requests in IDYLL normalized to the total latency of demand TLB miss requests in the baseline execution (the lower the better). As observed, the total demand TLB miss request latency of our approach is reduced by about 60% compared to the baseline. The reductions in demand TLB miss request latency directly translate to performance improvements. For example, for `PR` and `IM`, the total demand TLB miss request latency of our approach is only about 25% of the baseline, thus achieving a significant performance improvement.

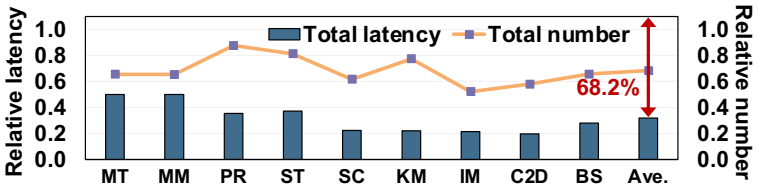

Figure 56: Total number of invalidation requests and total latency of invalidation requests.

Invalidation requests: The bar in Figure 56 shows the total latency of invalidation requests in IDYLL normalized to the total latency of invalidation requests in the baseline execution, and the line in the figure shows the percentage of invalidation requests in IDYLL normalized to the baseline. First, our approach eliminates all unnecessary requests so that the average number of invalidation requests in IDYLL is reduced by 32% of the baseline. The total invalid request latency in our approach is reduced by 68.2% compared to the baseline. This is because (i) the total number of invalidation requests is reduced and (ii) we coalesce multiple invalidation requests for page table walks, which can share the same page walk cache entries, thereby improving the L2 level page walk cache hit rate of invalidation requests and reducing the latency caused by page walk cache misses.

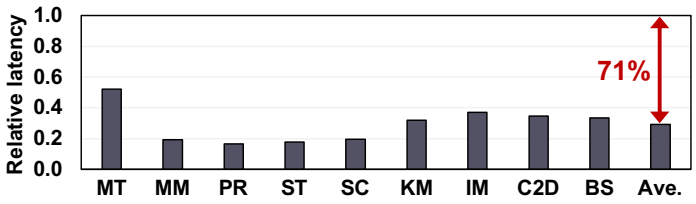

Figure 57: Page migration waiting latency.

Page migration waiting latency: Figure 57 shows the total latency of page migration waiting in IDYLL normalized to the total latency of page migration waiting in the baseline execution. The results indicate an average of 71% latency reduction brought by our approach compared to the baseline. Recall that, in the baseline, page migration cannot start before (i) all the GPUs finish page table walks for the invalidation requests, and (ii) the host finishes page table walks for the invalidation. While the GPU page table walk can happen concurrently with the host page table walk, the walking latency on the host side is expected to be much lower compared to the GPUs. This is because of the high bandwidth of the host page table walk and the fewer far faults that need to be handled by the host page table walk. As a result, our approach significantly reduces the page migration waiting latency as our approach only needs to perform the host-side page table walk to determine the GPUs with valid translations and register them in the IRMB of the corresponding GPU without performing page table walks in the GPU. We also want to emphasize that our approach does not affect the page data migration time as our focus is address translation invalidations. The figure only shows page migration waiting time which does not include the page data migrating time.

5.3.2 Sensitive Study

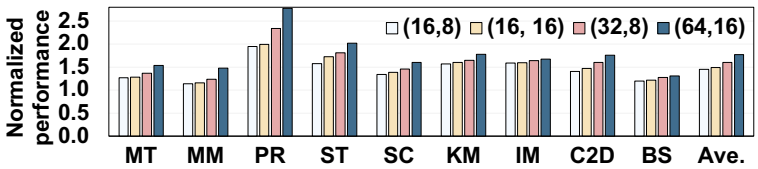

Figure 58: IDYLL with different IRMB size. The (x,y) on the legend indicates (size of bases, size of offsets).

IRMB size: Recall that, in our configuration of IDYLL, the IRMB is set to 32 different

bases with 16 offsets per base. In this study, we evaluate the performance impact with different IRMB configurations. The (x,y) indicates (size of bases, size of offsets). As shown in Figure 58, first, the performance improvement decreases as the IRMB size reduces. When IRMB size is reduced to (16, 8), the average performance improvement is 44.8% over the baseline, which is 25.1% lower than the default sizes (i.e., 32 bases and 16 offsets). This is because fewer invalidation requests can be kept with a small-sized IRMB, which leads to frequent IRMB eviction, introducing more contention to demand TLB miss requests. Second, when increasing the IRMB size to (64, 16), the average performance improvement is 76.9%, which is 7% higher than the default size. Considering the hardware overhead, we choose (32, 16) as our configuration.

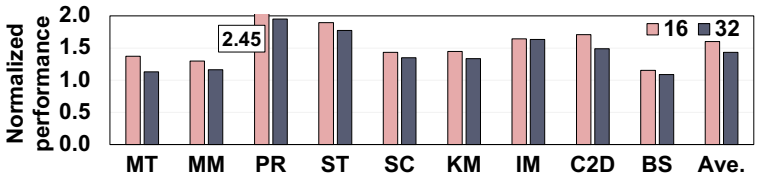

Figure 59: IDYLL with 16- and 32-threaded page table walk.

Number of page table walk threads: We evaluate IDYLL under different numbers of page table walk (PTW) threads in GMMU. Figure 59 shows the performance results normalized to the baseline execution with the same number of PTW threads. IDYLL achieves an average of 60% and 43.3% performance improvements with 16 and 32 GMMU PTW threads, respectively. Thus, with more PTW threads, the improvements remain significant with IDYLL, though the improvement slightly degrades as a large number of PTW threads reduces the contention and performance penalty caused by invalidations on demand TLB miss requests.

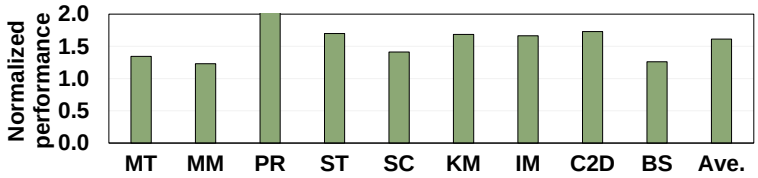

Figure 60: IDYLL with 2048-entry L2 TLB.

L2 TLB sizes: We evaluate IDYLL under a larger GPU L2 TLB size (2048 entries, 64-way, and 32-set). Figure 60 shows that IDYLL achieves 61.4% performance improvement

over the baseline using 2048-entry L2 TLB. Although a larger TLB size can keep more translations into the TLB and reduce the number of page table walks in the GMMU, the TLB shootdown caused by page migration makes a large TLB much less helpful.

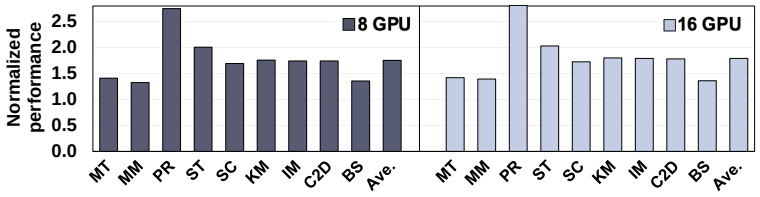

Figure 61: IDYLL with 8 and 16 GPUs.

Number of GPUs: We evaluate IDYLL in 8-GPU and 16-GPU systems. Figure 61 plots the performances of IDYLL with 8 GPUs and 16 GPUs normalized to the baselines with 8 GPUs and 16 GPUs. The average performance improvement of 8-GPU and 16-GPU is 75.3% and 79.1%, respectively. One can make the following observations. First, the performance improvement increases with more GPUs. Note that, for a fair comparison, we only increase the number of GPUs without changing the application's input dataset sizes. As a result, with more GPUs, the pages are more frequently shared across GPUs, and more page migrations are triggered, introducing more invalidation requests to each GPU. Our approach is effective in handling these significant invalidations by batch processing and lazy updates. Second, the trend of performance gains becomes slow as the number of GPUs increases. This is because, in our design, we map several GPU access bits to a single bit of the host-side PTE, which increases the false positive rate when sending invalidation requests to GPUs. However, the Lazy Invalidation mechanism still eliminates significant invalidation request interference to demand TLB miss requests. In a nutshell, IDYLL delivers performance improvements with more GPUs.

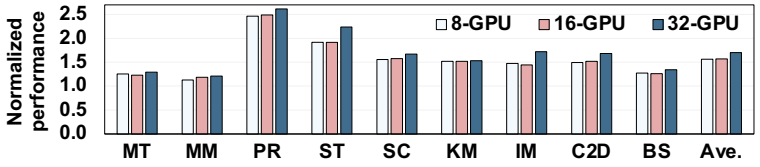

Figure 62: IDYLL with 4 unused bits and varying GPU count.

Number of unused bits: We next evaluate IDYLL with fewer unused bits (i.e., 4 unused bits). Figure 62 shows the performance of IDYLL under 8, 16, and 32 GPUs, using 4

unused bits. The results are normalized to the baseline execution with 8, 16, and 32 GPUs, respectively. Although the increased hash false positives when employing less number of unused bits can result in more unnecessary invalidation requests being sent to the GPU and potentially degrade the benefits one can obtain from In-PTE Directory Invalidation, our approach still achieves an average performance improvement of 56.5%, 57.1%, and 70.1%, respectively. This is mainly due to the effectiveness of Lazy Invalidation, which plays a significant role in enhancing the IDYLL performance.

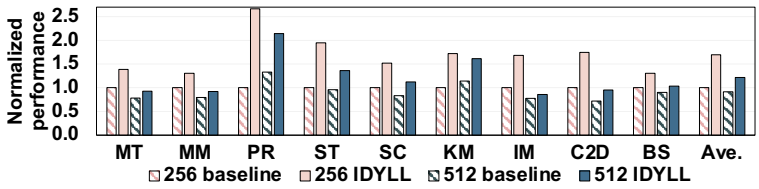

Figure 63: IDYLL with 512 access counter threshold.

Access counter threshold: In our discussion so far, we use the UVM default access counter threshold (256). We next evaluate the impact of a larger access counter threshold on IDYLL. Figure 63 shows the performances of baseline and IDYLL with a threshold of 512 (baseline-512 and IDYLL-512); the results are all normalized to the original baseline (baseline-256). One can make the following observations. First, IDYLL-512 outperforms baseline-512 by 30.0%. Second, this performance improvement is less than the improvement with a threshold of 256 (i.e., IDYLL-256 outperforms baseline-256 by 69.9%). This is because a higher threshold reduces the total number of page migrations, hence reducing the number of invalidations, making the potential for improvement less in our approach. However, we want to emphasize that having a larger access counter threshold does not necessarily guarantee better overall performance. Figure 63 also presents the baseline-512 performance normalized to the baseline-256. One can observe that the performance drops by 10% when using a threshold of 512. This is because a larger threshold leads to an increased number of remote accesses, exacerbating the NUMA overheads and causing a degradation in overall performance.

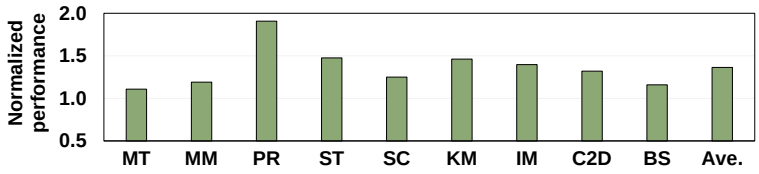

Figure 64: IDYLL with 2MB pages.

5.3.3 Adopting Large-sized Pages

In this study, we evaluate how IDYLL works with a 2MB large page. To sufficiently stress the virtual memory subsystem with 2MB pages, we enlarge the input sizes for each application. Figure 64 shows the performance improvement of IDYLL with 2MB page size normalized to the baseline execution with 2MB page size. The results indicate that IDYLL achieves an average of 36.3% performance improvement. It is expected that the gains drop from 4KB page size as the large page size increases TLB reach, page walk cache hit rate, and reduces the page walk contention. However, our approach remains effective, especially for those applications with substantial page sharing (such as `PR`). This is because adopting a large page size increases false sharing and still incurs a sizable amount of invalidation requests in the system.

5.3.4 Compared to Page Replication

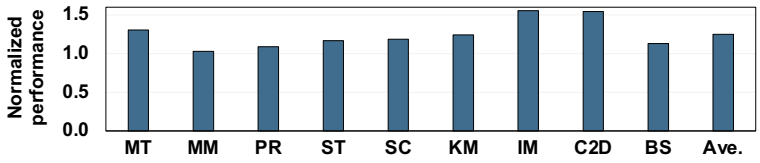

Figure 65: IDYLL with page replication.

We next compare IDYLL with page replication. Different from access counter-based remote mapping, page replication allows the page to be duplicated among GPUs so read accesses do not require page migration. It consumes more physical memory space but avoids NUMA overheads as in access counter-based remote mapping. Figure 65 shows the performance of IDYLL normalized to the page replication. IDYLL achieves an average of 25.0% performance improvement. Comparing the results with the performance of access counter-based page migration in Figure 54, the improvement is less, especially for `PR`, `ST`, and `SC`.

This is because these applications are read-intensive applications and the invalidation requests are significantly reduced, which makes less room for optimization. However, when a GPU performs a write/modification operation, the page replication approach coalesces all replications into a single page and each GPU needs to perform a page table walk to invalidate the corresponding PTE. Therefore, for write-intensive applications such as `IM` and `C2D`, our approach still significantly outperforms page replication. Note that, we do not simulate the over-subscription in this comparison experiment. Thus, with substantial sharing among multiple GPUs, page replication is not scalable and may decrease the overall performance.

6.0 Future Work

6.1 Design Option for IDYLL

In the event that the unused bits in page table entry are reserved for other purposes (e.g., implementing custom memory management policies, setting access permissions) [69, 38, 57], we propose an alternative in-memory directory design, , that tracks GPU translation residency for all pages in the system. The in-memory directory, referred to as the VM-Table, achieves the same functionality as the In-PTE Directory. Specifically, each entry in the VM-Table is 64 bits and stores VPN (45 bits) and GPU access bits (19 bits, all initialized to 0s). If the system has more than 19 GPUs, we employ the same hash function as the In-PTE Directory approach to hash the GPU access bits.

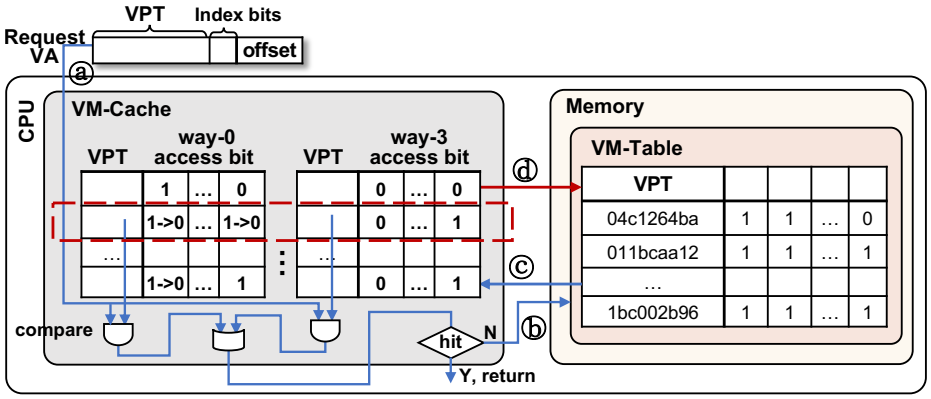

Figure 66: Overview of IDYLL-InMem.

Having every translation to access the in-memory VM-Table for page residency involves memory accesses that incur additional memory bandwidth and latency overheads. To mitigate the overheads, we propose a hardware-managed cache (VM-Cache) to cache frequently-accessed entries from the VM-Table. The VM-Cache has 64 entries (4-way associative) and uses write allocate and write back policy. Figure 66 illustrates our proposed design.

The execution flow is as follows. When UVM receives a page migration request, it looks up the VM-Cache to obtain the GPU access bits (ⓐ) and sends invalidations to those GPUs.

The lookup occurs in parallel to the host-side page table walk. If the entry is found in the VM-Cache, UVM sends invalidation requests based on the access bits. Meanwhile, all access bits, except for the bit of the GPU that initiated the page migration, are set to 0s in the VM-Cache. In contrast, on a VM-Cache miss, a memory access is generated to access the VM-Table (ⓑ). Two scenarios may happen. First, if the entry is found in the VM-Table, the corresponding entry is brought into the VM-Cache (ⓒ) and the access bits are updated. Second, if the entry is not found in the VM-Table, which can occur only when the page is first accessed by a GPU, the entry is registered in the VM-Cache. We use LRU replacement for the VM-Cache where evicted entries are written back to the VM-Table (ⓓ). In the case of far faults, the UVM driver performs a page table walk to get the translation mapping and, at the same time, checks the VM-Cache to update the GPU access bits. The checking process is similar to the lookup process discussed above, except for updating the corresponding GPU access bit to 1 in the entry.

6.2 MIG-TLB Optimization

Multi-GPUs are engineered to handle tasks that require high throughput and parallel processing, which makes them ideal for the training phase of machine learning applications. However, during the inference phase, the full potential of GPUs may not be tapped into due to the typically lower computational requirements, resulting in underutilization of GPU resources. To address this inefficiency and optimize resource usage, many cloud services have turned to deploying clusters of GPUs, applying multi-tenancy techniques to maximize throughput [54, 2, 83]. Multi-tenancy allows multiple users or tasks to share the same physical GPU resources. NVIDIA's Multi-Instance GPU (MIG) [51] technology is one of the prominent GPU-sharing approaches. MIG enables a single physical GPU to be divided into several isolated instances, each with its own set of resources, including streaming multiprocessors (SMs), memory, and caches. NVIDIA MIG is designed to offer complete isolation of resources for each instance, ensuring performance without interference from other instances. However, a recent study [89] has indicated that while MIG effectively partitions most of the

memory system, it does not provide partitioning to the last-level TLB (i.e., L3 TLB), which leads to multi-tenant contend for shared L3 TLB.

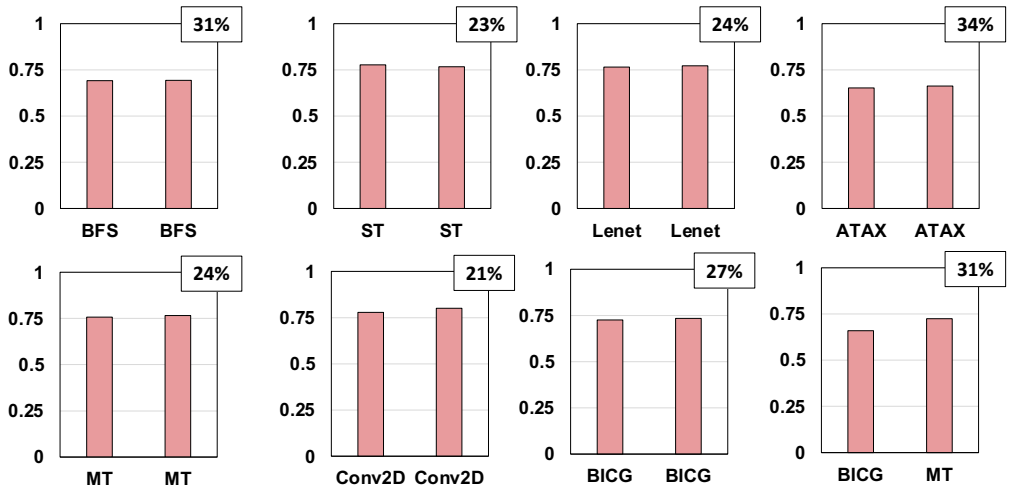

Figure 67: Performance of GPU applications with 3g: 3g instances.

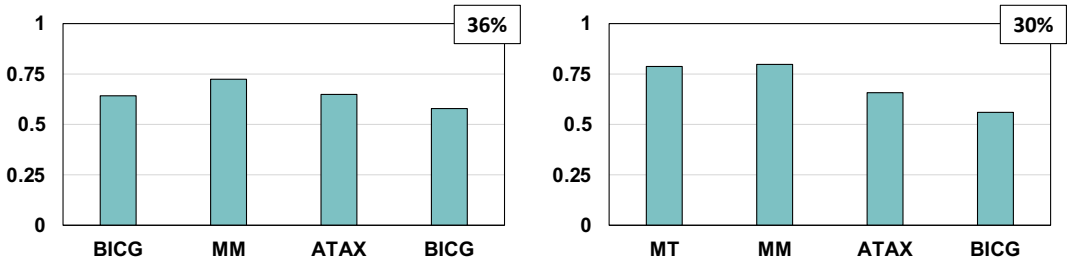

Figure 68: Performance of GPU applications with 3g: 2g: 1g: 1g instances.

To quantify the performance impact caused by interference and contention at L3 TLB, we run representative applications on an NVIDIA A100 GPU with MIG-enabled. The whole GPU is partitioned into different-sized instances, e.g., 3g: 3g and 3g: 2g: 1g: 1g. Figure 67 and Figure 68 show the weighted speedup of the different workload combinations. One can make the following observations. First, L3 TLB contention degrades individual application's performance. For example, in `ST-ST` workload, the performance drops 23% on average. Second, the performance drop increases with the increase in the number of concurrently running applications. For example, in `BICG-BICG` workload, the performance drop of `BICG` is 27%. However, in `BICG-MM-ATAX-BICG` workload, with the same instance size (i.e., 3g), the performance drop of `BICG` is 36%. This is because diverse workloads increase the TLB thrashing, where more applications compete for limited-size TLB and lead to frequent evictions of TLB

entries.

Table 7: Application workloads.

Abbr.	MPKI	Category
FIR	0.32	L
FFT	0.48	L
ST	21.92	M
NW	38.41	M
CONV2D	1.92	M
MT	205.33	H
ATAX	204.76	H
BICG	208.97	H

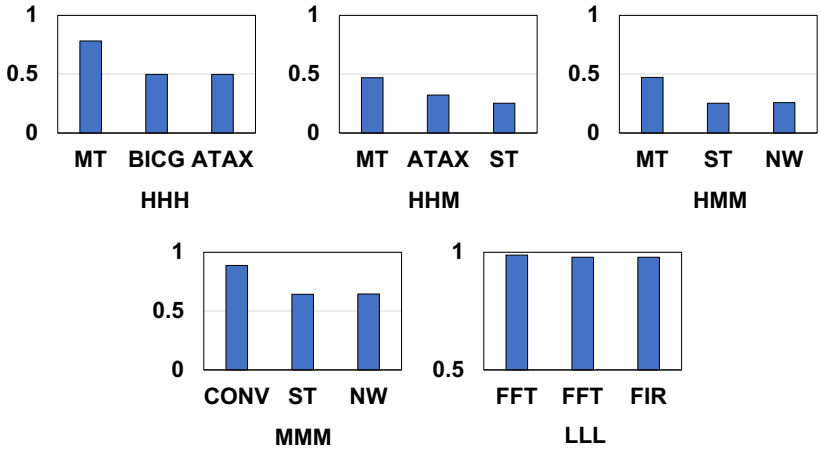

Figure 69: Performance of GPU applications with 3g: 2g: 2g instances.

Figure 69 presents a simulation-based study that depicts the weighted performance of each application in the workload when co-running. We use MGPUsim to model different sizes of instances. Table 7 shows the workloads where each workload contains three applications. The multi-application workloads are formed by characterizing their memory access intensity. Specifically, we quantify each application's misses-per-kilo-instructions (MPKI) of the address translations at L2 TLB. Based on the L2 TLB miss MPKI, we classify the applications into three categories: Low (L, MPKI<1), Medium (M, 1<MPKI<50), and High (H, MPKI>100). The workloads are formed as a mix of applications from different categories, including LLL, MMM, MMH, MHH, and HHH. One can make the same observation that L3 TLB contention degrades individual application's performance. Specifically, for applications with higher MPKI, the impact is more significant. For example, in `FFT-FFT-FIR` workload, the performance impact of each application is relatively minor. While in `MT-ATAX-BICG` workload, performance drops by 41% on average.

In A100 GPU, there are 16 sub-entries in one L2 TLB and L3 TLB entry, and they have a one-to-one mapping relationship with the address translations for 16 pages of size 64KB located in the same 1MB-aligned range. If any sub-entry encounters an eviction, the rest of them are also invalidated. We then study the utilization of sub-entries before it evicted.

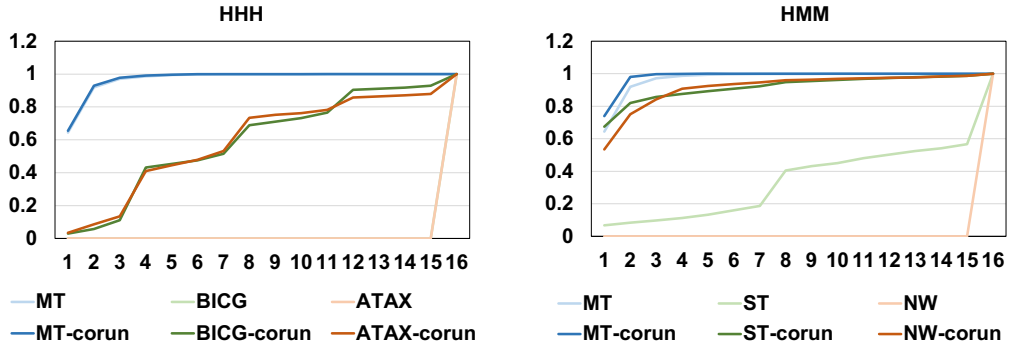

Figure 70: Cumulative distribution function (CDF) of sub-entry utilization.

Figure 70 shows the Cumulative Distribution Function (CDF) of sub-entry utilization of HHH and HMM application combinations when run alone and when run alongside other applications. One can make the following observations. First, there exist substantial underutilized TLB entries. For example, in `MT`, 65% of TLB entries are evicted where only one sub-entry is used. Second, the underutilization becomes more server in scenarios where applications are run alongside others. A greater number of TLB entries are evicted with only one sub-entry occupied. For example, for application `ATAX`, all TLB entries are evicted with fully utilized when running alone, while 50% TLB entries are evicted with 6-entry occupied when co-running with `MT` and `BICG`. This is because each application competes for a limited number of TLB entries, resulting in interference that leads to frequent evictions.

To reduce contention for shared L3 TLB entries and mitigate underutilization here multiple applications run concurrently, we will propose dynamic TLB partitioning and the sharing of sub-entries techniques. Specifically, first, we will propose real-time monitoring mechanisms to assess application-specific TLB usage patterns and workload characteristics. We will develop adaptive algorithms capable of dynamically resizing TLB partitions in response to the observed demand, ensuring applications with higher TLB needs are allocated a proportionally larger share of TLB resources. Second, we allow applications that access the same virtual addresses but under different process IDs to share TLB sub-entries to maximize the

utilization of existing TLB entries.

6.3 Disaggreagated Page Table

Scalability is limited by latency. Recall that MMU-managed multi-GPUs employ "remote mapping" to reduce frequent GPU local page faults. However, it increases address translation latency when shared pages migrate among GPUs. Specifically, when a page migrates, the translations, including those remote mappings in each GPU, need to be invalidated and updated. The process includes the following steps: i) The invalidation message is broadcast to all GPUs. ii) Each GPU, upon receiving an invalidation message, performs a local page table walk for the translation and, if found, invalidates the translation. In this process, those page table walks for invalidation and update cause significant contention of GPU page table walk threads. Moreover, the broadcasting can cause interconnect congestion which delays the subsequent translation requests.

Application	Footprint (GB)	Page Table Size (GB)	GPU Memory %		
			16 GB	32 GB	80 GB
Graph500	1280	2.56	16%	8%	3.2%
XSBench	1375	2.75	17%	8.5%	3.4%
GPT3	700	1.4	8.75%	4.38%	1.75%
Gshard	2400	4.8	30%	15%	6%

Figure 71: Memory footprint and page table size of cloud applications.

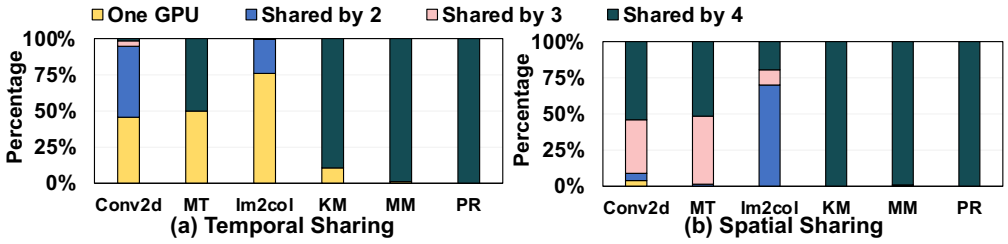

Figure 72: Page table page sharing across GPUs.

Scalability is limited by space. The memory for bookkeeping the page table (i.e., page table pages (PT pages)) is another scalability constraint. Table 71 lists several representative cloud applications and their memory footprints. As one can observe, the page

table can occupy up to 30% of the GPU memory. The occupied space also increases with more GPUs as each GPU keeps its page table separately in its local memory. Moreover, page sharing across multiple GPUs leads to the same PT page being allocated in different GPUs' local memories. The more pages are shared across GPUs, the more memory space is occupied. Such sharing can be divided into two bodies: *temporal* sharing and *spatial* sharing. In temporal sharing, the same translation is accessed by different GPUs during execution. In contrast, spatial sharing refers to the scenario where different GPUs access different translations in the same PT page. For example, assuming 8 bytes per page table entry (PTE), each 4KB PT page can hold 512 PTEs. If two GPUs access different PTEs within the 512 PTEs, the 4KB PT page will be allocated on both GPUs. As a preliminary study, Figure 72 shows the results of kernels used in cloud applications (e.g., `MM` and `CONV2D` are used in deep learning inference and training). We observe that there is an average of 92% PT pages being shared by multiple GPUs, and there is an average of 71.2% PT pages being shared by four GPUs.

To improve GPU scalability, we propose *disaggregated page table*. The fundamental idea is to reduce the number of duplicated and redundant address translations across GPUs and store only *one* copy of each translation across GPUs. While employing disaggregated page table reduces the memory space occupied by page tables, its design comes with overheads. First, translation requests may need to go remote owner GPU instead of local lookup since the page table is partitioned. Second, when a data page migrates, say from GPU_0 to GPU_1 as an example, it can happen that the translation owner is GPU_2. As a result, the migration will cause a page table update in GPU_2 to ensure the page table is coherent. Such update requests may cause contention in GPU_2, thereby delaying those translations on the execution critical path in GPU_2. We will also propose **PT page migration** to address the above overheads.

Bibliography

[1] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 607–618, 2015.

[2] Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. Multi-tenancy in cloud computing. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 344–351, 2014.

[3] Tyler Allen and Rong Ge. Demystifying gpu uvm cost with deep runtime and workload analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 141–150, 2021.

[4] Tyler Allen and Rong Ge. 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*, pages 1–15, 2021.

[5] AMD. AMD APP SDK OpenCL Optimization Guide, 2015.

[6] AMD. AMD Radeon R9 Series Gaming Graphics Cards with High- Bandwidth Memory., 2015.

[7] AMD Corp. I/O Virtualization Technology(IOMMU) Specification, 2016.

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

[9] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. Mosaic: A gpu memory manager with application-transparent support for multiple page sizes. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 136–150, Oct 2017.

[10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 48–59, New York, NY, USA, 2010. ACM.

[11] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli. Griffin: Hardware-software support for efficient page migration in multi-gpu systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 596–609, 2020.

[12] Trinayan Baruah, Yifan Sun, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. Valkyrie: Leveraging inter-tlb locality to enhance gpu performance. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 455–466, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[14] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: Asynchronous multi-gpu programming model with applications to large-scale graph processing. *ACM Transactions on Parallel Computing (TOPC)*, 7(3):1–27, 2020.

[15] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level tlbs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 62–63, 2011.

[16] A. Bhattacharjee, D. Lustig, and M. Martonosi. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers, 2017.

[17] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 383–394, New York, NY, USA, 2013. ACM.

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

[19] Abhishek Bhattacharjee. *Appendix L:Advanced Concepts on Address Translation*. Elsevier, 2019.

[20] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Citeseer, 2004.

[21] Steven Chien, Ivy Peng, and Stefano Markidis. Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57. IEEE, 2019.

[22] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler. Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 926–939, 2020.

[23] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU-3: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[24] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.

[25] Shi Dong and David Kaeli. Dnnmark: A deep neural network benchmark suite for gpus. In *Proceedings of the General Purpose GPUs*, pages 63–72. 2017.

[26] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, June 2015.

[27] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461. IEEE, 2020.

[29] Timothy D.R. Hartley, Umit Catalyurek, Antonio Ruiz, Francisco Igual, Rafael Mayo, and Manuel Ujaldon. Biomedical image analysis on a cooperative cluster of gpus and multicores. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 413–423, New York, NY, USA, 2014. ACM.

[30] Bongjoon Hyun, Youngeun Kwon, Yujeong Choi, John Kim, and Minsoo Rhu. Neummu: Architectural support for efficient address translations in neural processing units. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1109–1124, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Intel. 5-Level Paging and 5-Level EPT, 2017.

[32] Intel. The Future of Core, Intel GPUs, 10nm, and Hybrid x86, 2018.

[33] J. Andrew Rogers. MetroHash: Faster, Better Hash Functions, 2015.

[34] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive insertion policies for managing shared caches. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 208–219, 2008.

[35] JEDEC. High bandwidth memory (hbm) dram 2. *Jesd235*, 2020.

[36] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, page 223–234, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 66–78, New York, NY, USA, 2015. ACM.

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

[39] Bingyao Li, Jieming Yin, Anup Holey, Youtao Zhang, Jun Yang, and Xulong Tang. Trans-FW: Short Circuiting Page Table Walk in Multi-GPU Systems via Remote Forwarding. In *Proceedings of the 29rd International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[40] Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. Improving address translation in multi-gpus via sharing and spilling aware tlb design. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1154–1168, 2021.

[41] Xinjian Long, Xiangyang Gong, and Huiyang Zhou. Deep learning based data prefetching in cpu-gpu unified virtual memory. *arXiv preprint arXiv:2203.12672*, 2022.

[42] Xinjian Long, Xiangyang Gong, and Huiyang Zhou. An intelligent framework for oversubscription management in cpu-gpu unified memory. *arXiv preprint arXiv:2204.02974*, 2022.

[43] Zhulin Ma, Yujuan Tan, Hong Jiang, Zhichao Yan, Duo Liu, Xianzhang Chen, Qingfeng Zhuge, Edwin Hsing-Mean Sha, and Chengliang Wang. Unified-tp: A unified tlb and page table cache structure for efficient address translation. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 255–262. IEEE, 2020.

[44] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 1023–1036, New York, NY, USA, 2019. ACM.

[45] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. Beyond the socket: Numa-aware gpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–135, 2017.

[46] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[47] Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. Gps: A global publish-subscribe model for multi-gpu memory management. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–58, 2021.

[48] Nikolay Sakharnykh. Unified Memory on Pascal and Volta, 2017.

[49] NVIDIA. DB2 Launch Datasheet Deep Learning Letter WEB, 2018.

[50] NVIDIA. NVIDIA Linux Open GPU Kernel Module Source, 2022.

[51] NVIDIA. Nvidia driver documentation - nvidia multi-instance gpu user guide, 2023.

[52] NVIDIA Corp. Everything you need to know about unified memory, 2018.

[53] NVIDIA Corp. NVIDIA A100 Tensor Core GPU Architecture, 2020.

[54] Isaac Odun-Ayo, Sanjay Misra, Olusola Abayomi-Alli, and Olasupo Ajayi. Cloud multi-tenancy: Issues and developments. In *Companion Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17 Companion, page 209–214, New York, NY, USA, 2017. Association for Computing Machinery.

[55] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border control: Sandboxing accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 470–481, New York, NY, USA, 2015. Association for Computing Machinery.

[56] M. Parasar, A. Bhattacharjee, and T. Krishna. Seesaw: Using superpages to improve vipt caches. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 193–206, 2018.

[57] C. H. Park, T. Heo, J. Jeong, and J. Huh. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 444–456, June 2017.

[58] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Every walk's a hit: making page walks single-access cache hits. In *Proceedings of*

the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pages 128–141, 2022.

[59] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing tlb reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 558–567, Feb 2014.

[60] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Dec 2015.

[61] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 258–269, USA, 2012. IEEE Computer Society.

[62] B Pratheek, Neha Jawalkar, and Arkaprava Basu. Improving gpu multi-tenancy with page walk stealing. In *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

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

[64] Jihyun Ryoo, Mengran Fan, Xulong Tang, Huaipan Jiang, Meena Arunachalam, Sharada Naveen, and Mahmut T Kandemir. Architecture-centric bottleneck analysis for deep neural network applications. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 205–214. IEEE, 2019.

[65] Albert Segura, Jose-Maria Arnau, and Antonio González. Scu: A gpu stream compaction unit for graph processing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 424–435, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–35, 2018.

[67] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu. Scheduling page table walks for irregular gpu applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 180–192, June 2018.

[68] S. Shin, M. LeBeane, Y. Solihin, and A. Basu. Neighborhood-aware address translation for irregular gpu applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 352–363, 2018.

[69] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 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*, pages 1093–1108, 2020.

[70] JEDEC Standard. High bandwidth memory (hbm) dram. *Jesd235*, 2013.

[71] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[72] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 197–209, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Rafael Ubal, Xiang Gong, Shane Treadway, Yuhui Bao, Vincent Zhao, José L Abellán, et al. Mgsim+ mgmark: A framework for multi-gpu system research. *arXiv preprint arXiv:1811.02884*, 2018.

[74] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut Kandemir, and Chita Das. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture (HPCA)*, 2017.

[75] Xulong Tang, Ziyu Zhang, Weizheng Xu, Mahmut Taylan Kandemir, Rami Melhem, and Jun Yang. Enhancing address translations in throughput processors via compression. In *Proceedings of the ACM International Conference on Parallel Architectures*

and Compilation Techniques, PACT '20, page 191–204, New York, NY, USA, 2020. Association for Computing Machinery.

[76] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *2008 International Symposium on Computer Architecture*, pages 51–62, 2008.

[77] Tyler Allen. UVM performance evaluation, 2023.

[78] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A Jiménez, and Marc Casas. Exploiting page table locality for agile tlb prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 85–98. IEEE, 2021.

[79] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171, April 2016.

[80] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[81] Jinhui Wei, Jianzhuang Lu, Qi Yu, Chen Li, and Yunping Zhao. Dynamic gmmu bypass for address translation in multi-gpu systems. In Xin He, En Shao, and Guangming Tan, editors, *Network and Parallel Computing*, pages 147–158, Cham, 2021. Springer International Publishing.

[82] Jinhui Wei, Jianzhuang Lu, Qi Yu, Chen Li, and Yunping Zhao. Dynamic gmmu bypass for address translation in multi-gpu systems. EasyChair Preprint no. 4179, EasyChair, 2020.

[83] Katie Wood and Mark Anderson. Understanding the complexity surrounding multitenancy in cloud computing. In *2011 IEEE 8th International Conference on e-Business Engineering*, pages 119–124, 2011.

[84] Chenhao Xie, Fu Xin, Mingsong Chen, and Shuaiwen Leon Song. Oo-vr: Numa friendly object-oriented vr rendering framework for future numa-based multi-gpu sys-

tems. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 53–65, New York, NY, USA, 2019. Association for Computing Machinery.

[85] Yuejian Xie and Gabriel H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 174–183, New York, NY, USA, 2009. Association for Computing Machinery.

[86] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee. Hardware translation coherence for virtualized systems. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 430–443, 2017.

[87] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 698–710, New York, NY, USA, 2019. ACM.

[88] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. Combining hw/sw mechanisms to improve numa performance of multi-gpu systems. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 339–351. IEEE, 2018.

[89] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. T unne l s for b ootlegging: Fully reverse-engineering gpu tlbs for challenging isolation guarantees of nvidia mig. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 960–974, 2023.