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

Bingyao Li
University of Pittsburgh
Pittsburgh, PA, USA
bil35@pitt.edu

Jieming Yin
Lehigh University
Bethlehem, PA, USA
yin@lehigh.edu

Youtao Zhang
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Xulong Tang
University of Pittsburgh
Pittsburgh, PA, USA
tax6@pitt.edu

ABSTRACT

In recent years, the ever-growing application complexity and input dataset sizes have driven the popularity of multi-GPU systems as a desirable computing platform for many application domains. While employing multiple GPUs intuitively exposes substantial parallelism for the application acceleration, the delivered performance rarely scales with the number of GPUs. One of the major challenges behind is the address translation efficiency. Many prior works focus on CPUs or single GPU execution scenarios while the address translation in multi-GPU systems receives little attention. In this paper, we conduct a comprehensive investigation of the address translation efficiency in both "single-application-multi-GPU" and "multi-application-multi-GPU" execution paradigms. Based on our observations, we propose a new TLB hierarchy design, called *least-TLB*, tailored for multi-GPU systems and effectively improves the TLB performance with minimal hardware overheads. Experimental results on 9 single-application workloads and 10 multi-application workloads indicate the proposed least-TLB improves the performances, on average, by 23.5% and 16.3%, respectively.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Virtual memory**.

KEYWORDS

multi-application; multi-GPU; TLB

ACM Reference Format:
Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. 2021. Improving Address Translation in Multi-GPUs via Sharing and Spilling aware TLB Design. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480083

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480083

1 INTRODUCTION

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 [60, 79], graph processing [30, 61, 63], scientific computing [47, 73], and VR/AR [27, 82]. Such popularity has motivated a comprehensive literature of GPU optimizations focusing on "single-application-single-GPU" and "multi-application-single-GPU" execution paradigms [12, 15, 24, 64, 65]. 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 the application execution characteristics, GPU vendors have explored different incarnations of multi-GPU systems, including NVIDIA's DGX [48] and Intel Xe [32]. While these platforms improve the system parallelism, their potential and the delivered application performances are constrained by several design challenges [39, 42]. In particular, an address translation lookup that misses the GPU local TLB hierarchy 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 [14, 17, 41]. In our study, we observe a significant performance drop due to IOMMU TLB contention.

Many prior works have explored TLB optimizations in CPUs and GPUs. Table 1 summarizes the address translation optimizations in the literature. We also list the features as the columns in the table. '-' represents that the optimization is less effective or is difficult to be extended to support the features, whereas '+' represents the optimization works well supporting the features. Most of these optimizations, however, focus on single-GPU/CPU executions and cannot be effectively applied to the multi-GPU environment. First, employing large pages improves TLB performance but suffers from

severe intra-page fragmentation [8, 51, 56]. Such fragmentation causes frequent page swapping between CPU and GPU due to the limited GPU memory capacity. Moreover, applications with irregular memory access patterns (e.g., graph applications) benefit little from larger pages (we quantitatively discuss large pages in Section 5.4). Second, range-TLB [41, 52, 84], cluster-TLB [54, 55], and TLB-compression techniques [75] rely on continuous and stride 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. Third, TLB speculation and prediction [11] rely on predictable access patterns to deliver high accuracy. However, such predictable patterns are rarely observed in IOMMU TLB especially when multiple applications interfere with each other. The prediction accuracy can reduce dramatically and lead to performance degradation. Fourth, different from the local TLB hierarchy within a single GPU where the L1 TLB and L2 TLB (if any) have comparable lower and constant access latencies, accessing the IOMMU TLB from each GPU experiences much longer latency and has large variance among the access latencies. As a result, prior works (e.g., [13]) that focus on local TLB hierarchical are less effective when applied to multi-GPU execution. This is because these approaches rely on fast inquires and responses among the local and remote TLBs, which is not possible when the remote TLB is in another GPU or in CPU. Finally, several prior works focus on optimizing page table walk (PTW) in single GPUs [59, 64, 65]. While some of these works can be extend to support multi-application and multi-GPU systems, they do not optimize the TLB performance and can be combined with TLB optimizations to achieve further improvements (discussed in Section 5.6). These limitations of the prior works motivate us to rethink the TLB designs in the context of multi-GPU environment:

> *Can we orchestrate all levels of TLBs in a holistic manner to achieve better performance?*

In this paper, we target IOMMU 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 [19, 85] 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. To the best of our knowledge, this paper conducts the first investigation and exploration on multi-GPU TLB hierarchy. The paper makes the following major contributions:

- We comprehensively investigate the TLB performance in multi-GPU systems under both single- and multi-application execution. We observe that the "remote" IOMMU TLB suffers from severe thrashing compared to the "local" GPU TLBs. Such thrashing causes few address translation reuses to be captured by the IOMMU TLB, leading to an average of 42.3% and 29% performance drop for single- and multi-application execution, respectively.
- We propose least-TLB to mitigate TLB thrashing. Specifically, least-TLB consists of three major components to coordinate all levels of TLBs in a multi-GPU system. First, a least-inclusive TLB hierarchy is proposed to improve the TLB reach and TLB hit rate by reducing translation redundancy. Second, we propose a spilling mechanism to reduce the IOMMU TLB contention in multi-application execution. Third, we design a Local TLB Tacker to track the entries of L2 TLB with the minimal hardware overhead.
- We evaluate least-TLB design using 9 single-application workloads and 10 multi-application workloads. Experimental results show that least-TLB significantly improves TLB reach and TLB hit rate, resulting in an average of 23.5% and 16.3% performance improvement for single- and multi-application workloads, respectively.

Table 1: Comparison with prior techniques.

Techniques	Irregular accesses	Stride accesses	No internal fragmentation	Support Multi-GPU	Support Multi-App
Range TLB [41, 52, 84]	–	–	+	–	–
Cluster TLB [54, 55]	–	–	+	–	–
Large page [8, 51, 56]	–	–	–	–	–
Eager paging [14, 29]	–	+	–	–	–
Speculative TLB [11]	–	+	+	–	–
Probing TLB [13]	–	+	+	–	–
TLB Compression [75]	+	+	+	–	–
irregular-app PTW [64, 65]	+	+	+	–	–
multi-tenancy PTW [59]	+	+	+	–	+
Our work (least-TLB)	+	+	+	+	+

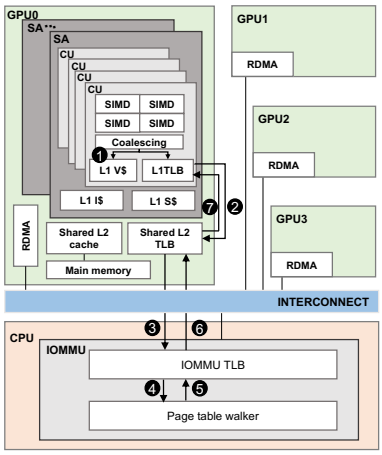

Figure 1: Baseline GPU architecture.

2 BACKGROUND

2.1 GPU Architecture

We target discrete multi-GPU system and Figure 1 shows the baseline system organization. The GPUs are modeled based on AMD GCN architecture [5]. A GPU consists of multiple shader arrays (SAs), and each SA consists of four compute units (CUs)[1]. Within each CU there are four SIMD units, where each SIMD unit comprises 16 processing elements (PEs), capable of handling 16 work-items in parallel. Each CU has a private L1 data cache (L1V$). All CUs within an SA share the L1 scalar cache (L1S$) and the L1 instruction cache (L1I$). All CUs across SAs share the L2 cache, which is connected to

[1]In this paper, we use CU in AMD terminology. A CU is equivalent to a streaming multiprocessor (SM) in NVIDIA terminology.

the GPU main memory. Apart from the cache hierarchy, each GPU also includes multi-level TLBs for address translation. Specifically, each CU has a fully associative private L1 TLB. A larger L2 TLB is shared among all CUs with a GPU. All the GPUs are connected with high-bandwidth connections (e.g., InfiniBand [53]), and they are connected to the CPU. 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 [20, 50, 81]. 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).

2.2 Address Translation in GPUs

Figure 1 also illustrates the 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 [12, 68]. 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 [6, 20].

Unlike our targeted discrete multi-GPU systems, there are other multi-GPU systems where each GPU implements its own local page tables [49]. That is, upon a lookup misses in the L2 TLB, the request is forwarded to the local page table walker. Only the local page faults are forwarded to IOMMU. We also evaluate the proposed least-TLB under such type of multi-GPU systems and present the quantitative results in Section 5.3.

It is important to note that, there exist different policies to manage multi-level TLBs, such as "strictly-inclusive", "mostly-inclusive", and "exclusive" [18, 33, 36]. Each policy has its pros and cons. For instance, the strict-inclusive policy facilitates translation sharing in shared TLB, but requires invalidation in private TLBs when an entry in the shared TLB is evicted. Similarly, exclusive design provides the best TLB reach but compromises the translation sharing. As the TLBs are read-only by applications, the mostly-inclusive TLB management is the most widely used one as it accommodates translation sharing while avoiding unnecessary invalidations [19, 85].

3 MOTIVATION

3.1 Baseline Configuration and Workloads

We use MGPUSim [68] 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 [4]. We heavily extended MGPUSim by adding the IOMMU module with a shared TLB to handle ATS and PPR requests (as we described in Section 2.2).

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 Baseline GPU configuration. In this paper, we target a 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. Actually, we provide a sensitivity study with eight GPUs in Section 5.3. Table 2 shows the baseline GPU configurations. The page size is set to 4KB[2]. 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 [18]. 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

3.1.2 Workloads. We select nine applications from AMDAPPSDK [3], Hetero-Mark [69], and SHOC [26] benchmark suites. These applications are listed in Table 3. 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

[2]We also evaluate our approach with large-sized pages in Section 5.4.

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.

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

Multi-application workload: To study the "multi-application-multi-GPU" execution, we use applications shown in Table 3 and add another application – Simple Convolution (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[3]. 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 [35, 59, 83]. 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 weight to the relative performance of each application [38]. 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]In our multi-application execution on four GPUs, each of the four applications occupies one GPU.

3.2 Single-Application-Multi-GPU

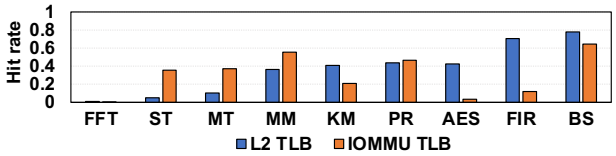

Figure 2: L2 TLB hit rate and IOMMU TLB hit rate in the baseline executions.

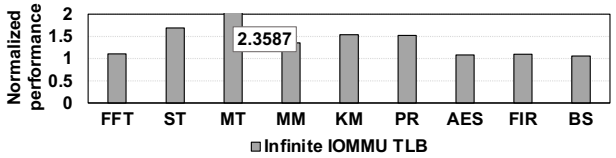

Figure 3: Normalized performance of infinite IOMMU TLB.

Observation 1: *Workload suffer from low TLB hit rate in both the L2 TLB and the shared IOMMU TLB.* Figure 2 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 3 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.

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 4. 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 5 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 the IOMMU TLB in the baseline execution. In Figure 6, 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,

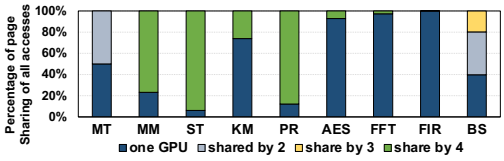

Figure 4: Percentage of page sharing.

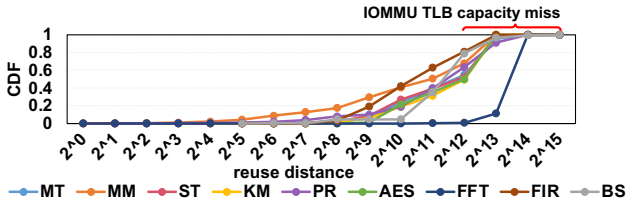

Figure 5: Cumulative distribution function (CDF) of translation reuse distances at the IOMMU TLB.

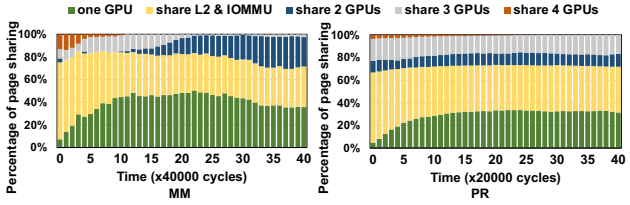

Figure 6: Page sharing during execution in MM and PR.

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.

3.3 Multi-Application-Multi-GPU

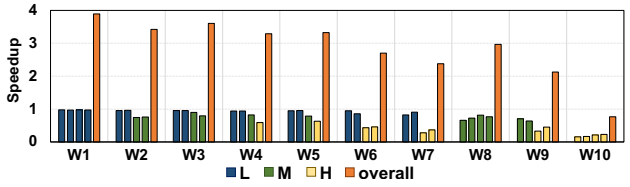

Figure 7: The speedup of each application in the workload and the overall weighted speedup of each workload.

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 7 shows the weighted speedup (defined in Section 3.1) of the ten workloads in Table 4. One can make the following observations. First, IOMMU TLB contention degrades individual application's 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

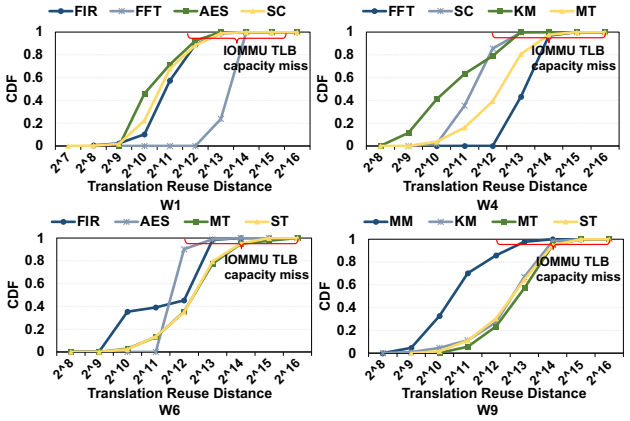

Figure 8: CDF of translation reuse distances in multi-application execution.

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 8 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.

4 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.

4.1 Single-Application-Multi-GPU

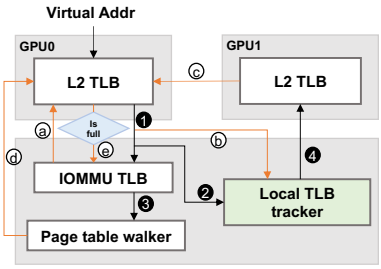

Figure 9: The lookup and insertion in least-TLB for single-application execution.

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 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 9) 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[28] in the IOMMU to *track* the translations in all GPUs' L2 TLBs. At a high-level, Cuckoo filter is similar to Bloom filters [23], 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 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 9 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.

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 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.

Lookup				
step	GPU	VAddr	Baseline	Least-TLB
1	0	0x5	miss	miss
2	1	0x1	miss	hit in IOMMU TLB
3	2	0x1	hit in IOMMU TLB	hit in remote GPU
4	3	0x1	hit in IOMMU TLB	hit in remote GPU

Insertion						
step	L2 TLB				IOMMU TLB	
	GPU0	GPU1	GPU2	GPU3	Baseline	Least-TLB
0	0x1	0x2	0x3	0x4	0x1 0x2 0x3 0x4	
1	0x5	0x2	0x3	0x4	0x2 0x3 0x4 0x5	0x1
2 - 4	0x5	0x1	0x1	0x1	0x3 0x4 0x5 0x1	0x2 0x3 0x4

Figure 10: A walk-through example for single-application execution.

A walk-through example: We use a simple example in Figure 10 to illustrate how least-TLB works. For simplicity, let us assume that each GPU L2 TLB has only one entry and IOMMU TLB has four entries. Initially, virtual pages 0x1-0x4 are present in the corresponding GPUs and the IOMMU is empty. In step 1, a request of 0x5 arrives at GPU_0 L2 TLB. Then, in least-TLB, 0x1 is evicted from L2 TLB and inserted into IOMMU TLB. In step 2, GPU_1 requests 0x1, which is hit in the IOMMU TLB. Then, the entry of 0x1 is removed from the IOMMU TLB and inserted into GPU_1 L2 TLB. In steps 3 and 4, GPU_2 and GPU_3 request 0x1 and they can hit in remote GPU_1. The figure also shows the baseline with most-inclusive TLB. As one can observe, least-TLB achieves better hit rate.

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.

4.2 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 3.3, 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 5.3 with different values of N.

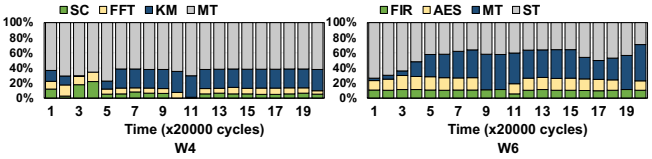

Figure 11: IOMMU TLB contents during executions of W4 and W6.

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

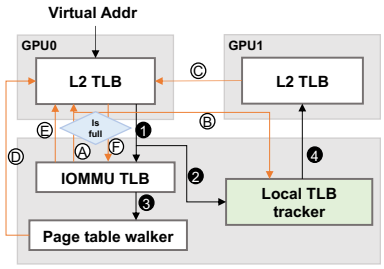

Figure 12: The lookup and insertion in least-TLB for multi-application execution.

Algorithm 2: Multi-applications Lookup & Insertion.

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

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

Lookup				
step	GPU	VAddr	Baseline	Least-TLB
1	2	0x11	miss	miss
2	2	0x7	miss	hit in remote GPU
3	1	0x12	miss	miss
4	1	0x13	miss	miss

	Spill					
L2 TLB	step	GPU0	GPU1	GPU2	GPU3	IOMMU_TLB
	0	0x1 0x2	0x3	0x4 0x5	0x6	0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE
Baseline	1	0x1 0x2	0x3	0x5 0x11	0x6	0x8 0x9 0xA 0xB 0xC 0xD 0xE 0x11
Least-TLB	1	0x1 0x2	0x3 0x7	0x5 0x11	0x6	0x8 0x9 0xA 0xB 0xC 0xD 0xE 0x4
Baseline	2	0x1 0x2	0x3	0x11 0x7	0x6	0x9 0xA 0xB 0xC 0xD 0xE 0x11 0x7
Least-TLB	2	0x1 0x2	0x3 0x8	0x11 0x7	0x6	0x9 0xA 0xB 0xC 0xD 0xE 0x4 0x5
Baseline	3	0x1 0x2	0x3 0x12	0x11 0x7	0x6	0xA 0xB 0xC 0xD 0xE 0x11 0x7 0x12
Least-TLB	3	0x1 0x2	0x8 0x12	0x11 0x7	0x6 0x9	0xA 0xB 0xC 0xD 0xE 0x4 0x5 0x3
Baseline	4	0x1 0x2	0x12 0x13	0x11 0x7	0x6	0xB 0xC 0xD 0xE 0x11 0x7 0x12 0x13
Least-TLB	4	0x1 0x2	0x12 0x13	0x11 0x7	0x6 0x9	0xA 0xB 0xC 0xD 0xE 0x4 0x5 0x3

Figure 13: A walk-through example for multi-application execution.

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).

A walk-through example: Figure 13 illustrates how least-TLB spilling works in multi-application execution. For simplicity, let us assume that each GPU L2 TLB has two entries and IOMMU TLB has eight entries. Initially, virtual pages 0x7-0xE are presented in the IOMMU TLB, where 0x7, 0x8 and 0xE are evicted from GPU_0 L2 TLB, 0x9 from GPU_1, 0xA-0xC from GPU_2, and 0xD from GPU_3. In the first step, a request of 0x11 arrives at GPU_2 L2 TLB. Then, in least-TLB, 0x4 is evicted from the L2 TLB and inserted into IOMMU TLB. The IOMMU TLB is also full so 0x7 is spilled into GPU_1, which has the fewest entries in the IOMMU TLB. In step 2, GPU_2 requests 0x7, which can hit in the remote GPU_1 L2 TLB. Then, 0x7 is inserted into GPU_2 L2 TLB and the entry is removed from the GPU_1 L2 TLB. The insertion of 0x7 causes the eviction of 0x5 from GPU_2. Then, 0x5 is inserted into the IOMMU TLB, and 0x8 is spilled to GPU_1. In step 4, GPU_1 requests 0x13, then 0x13 is inserted into GPU_1, and 0x8 is evicted without being inserted into the IOMMU TLB since it is a spilled entry. The figure also shows the hit rate in both baseline and least-TLB. Obviously, least-TLB has less number of misses.

4.3 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 [77] to estimate the area and power overheads of our approach. The result shows 0.19% area overhead compared to the area of IOMMU TLB.

4.4 Discussion

TLB shootdown: In the case of GPU local L1/L2 TLB shootdown, all TLB entries, including the spilled entries, are invalidated. As a result, when the tracker in the IOMMU sends a request to spilled entries, it cannot be found in remote TLBs. And this spilled entry will then be removed from the tracker. Note that this does not introduce

extra overheads because the IOMMU sends page table walk and the remote access simultaneously in our current implementation. In the case of IOMMU TLB shootdown, the tracker is also reset. Therefore, the spilled entries will be no longer accessible and will be eventually evicted from GPU local TLBs.

Limitations of least-TLB: Our approach increases the TLB hit rate relying on the increase in address translation locality. As such, our approach has limited improvements if i) the majority of the translation reuses distances are far beyond the IOMMU TLB capacity, or ii) the majority of the translation reuses distances are comparable short to be captured by the GPU local L1 and L2 TLBs.

Other types of system: So far, we model and evaluate the proposed least-TLB in the multi-GPU system where the page tables are shared on the CPU side. However, our least-TLB is not bound to a particular multi-GPU system. It also works for other types of multi-GPU systems where each GPU has its own page table (as we quantitatively evaluated in Section 5.3). For other types of accelerators, such as NPU [31], which requires high translation throughput in the IOMMU's PTWs, our least-TLB can potentially achieve better performance when combined with the optimizations on PTWs. This is because that the improved IOMMU TLB hit rate brought by least-TLB will reduce the traffic and contention on PTWs. Besides, we envision that the least-TLB will provide huge improvement potentials in future multi-chiplet heterogeneous systems where each GPU chiplet has limited TLB entries and does not have its own local memory. For the cases that heterogeneous devices are connected to the IOMMU, each device may have different local configurations (e.g., TLB sizes, QoS requirements, etc.). The proposed least-TLB can be enhanced to accommodate the fairness and efficiency requirements of different devices by adding the device IDs to the TLB entries. Then the least-TLB can perform device aware policies to manage the fairness and efficiency across heterogeneous devices. In our future work, we plan to investigate how the concept of least-TLB can be implemented in those systems.

5 EVALUATION

In this section, we evaluate the proposed least-TLB design using MGPUSim [68]. The system configuration is the same as the baseline shown in Table 2.

5.1 Single-Application-Multi-GPU Execution

Figure 14 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 3) 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.

We further look into the IOMMU TLB hit rate and remote GPU L2 TLB hit rate to understand the reason behinds the performance

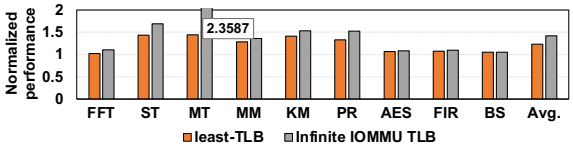

Figure 14: Normalized performance improvements in single-application execution.

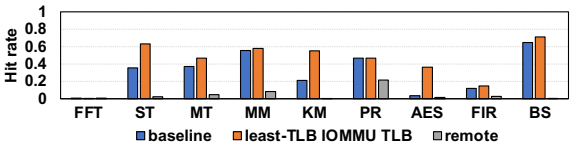

Figure 15: IOMMU TLB hit rate and remote hit rate in single-application execution.

improvements. As shown in Figure 15, 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 4), 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.

5.2 Multi-Application-Multi-GPU Execution

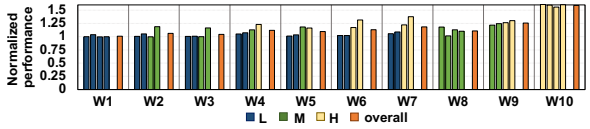

Figure 16: Normalized performance improvements in multi-application execution.

Figure 16 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 (as we discussed in Section 4.2). 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.

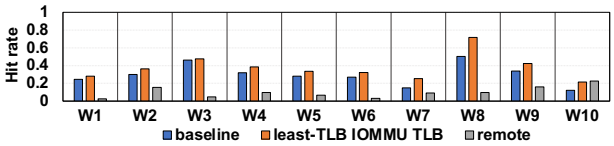

Figure 17: IOMMU TLB hit rate and remote hit rate in multi-application execution.

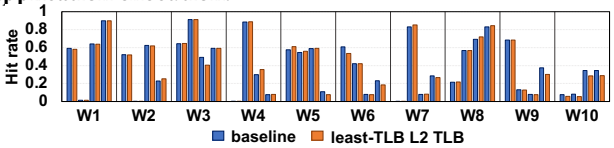

Figure 18: L2 hit rate in multi-application execution.

Figure 17 and Figure 18 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 17. 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 17), 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 base on the LRU replacement policy.

5.3 Sensitivity Study

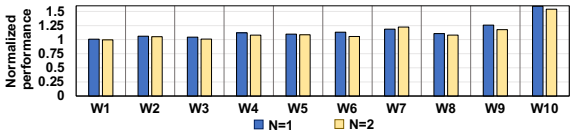

Figure 19: Normalized performance of spilling counter $N = 2$**.**

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 19 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 (described in Section 4.2), 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 [31]. 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 20 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 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 20 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 20. The CPU and GPUs are connected via PCIe (~300ns latency [76]) in our configuration, and the latency of HBM (~106.7ns latency [80]) 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 [67] and HBM2 [37]). On the contrary, the interconnection can be congested when multiple devices, especially heterogeneous ones with different quality of service (QoS) requirements, are connected to

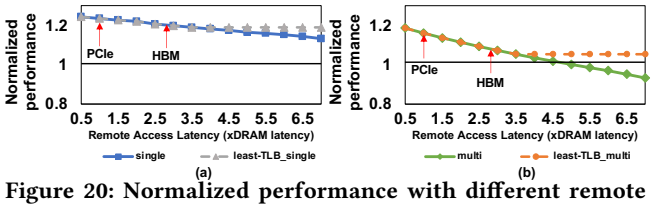

Figure 20: Normalized performance with different remote GPU access latencies. (a) single-application. (b) multi-application.

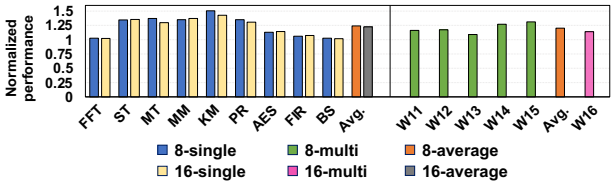

Figure 21: Normalized performance of least-TLB with 8 and 16 GPUs.

Table 5: 8- & 16-GPU multi-application workload.

Abbr.	Applications	Category
W11	AES, FIR, SC, PR, MM, KM, MT, ST	LLLMMMHH
W12	FIR, FFT, SC, MM, KM, MT, MT, ST	LLLMMHHH
W13	FIR, FFT, SC, AES, KM, MM, PR, BS	LLLLMMMM
W14	KM, MM, PR, BS, MT, MT, ST, ST	MMMMHHHH
W15	FIR, FFT, SC, AES, MT, MT, ST, ST	LLLLHHHH
W16	FIR, FFT, SC, AES, KM, MM, PR, BS MT, MT, ST, ST FIR, AES, KM, MT	LLLLLLMMMMMHHHHH

Table 6: Mix-workload.

Abbr.	Applications	Category
W17	FIR+KM, AES+MT, MM+ST	LM, LH, MH
W18	FIR+AES, KM+MM, MT+ST	LL, MM, HH
W19	SC+KM, FIR+MT, AES+ST	LM, LH, LH

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 20 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 GPUs to show its scalability. Figure 21 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 21 also shows the multi-application execution results. We use applications in Table 3 to form 5 8-GPU workloads and a 16-GPU workload as listed in Table 5. 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.

Sensitivity to the mix-workload per GPU: We evaluate least-TLB with mix-workload as shown in Table 6 where the application is not strictly one-to-one mapping to GPUs. Instead, two applications with different MPKIs run on the same GPU. Figure 22 shows that the performance of mix-workload execution improves by an average of 9.8% over the baseline execution. This demonstrates that our approach is general and can work under mixed workloads per GPU.

Sensitivity to GPUs with local page tables: We also model the multi-GPU system where each GPU has its own page table stored in the device memory and evaluate the proposed least-TLB. We

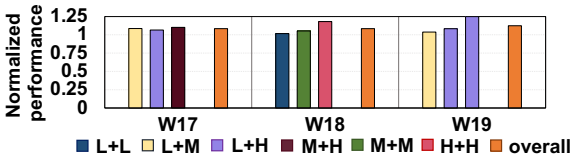

Figure 22: Normalized performance in mix-workload execution.

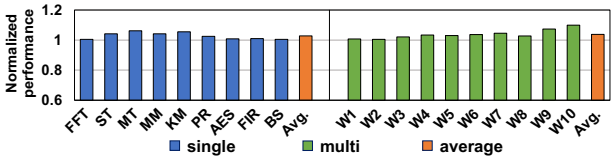

Figure 23: Normalized performance in multi-GPU systems where each GPU has its own page table.

implement the local page table for each GPU and model its address translation process (i.e., only local page faults are forwarded to IOMMU.) Figure 23 plots the performance of single-application and multi-application executions normalized to baseline executions using workloads in Table 3 and Table 4. The results show that least-TLB improves performance by 2.8% and 3.8% in single- and multi-application executions, respectively. The performance gain is less than that in AMD GCN-based systems. This is because page faults occur less frequently than L2 TLB misses.

5.4 Comparison to Large-sized Pages

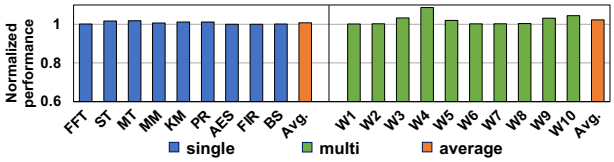

Figure 24: Normalized performance when using with 2MB page.

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 24 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.

5.5 Comparison to State-of-the-art

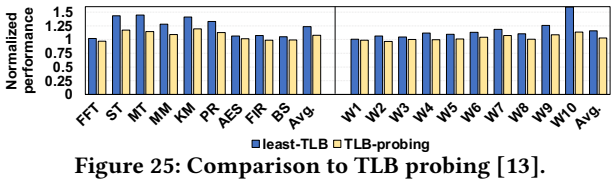

Figure 25: Comparison to TLB probing [13].

We compare least-TLB with the state-of-the-art TLB probing scheme [13]. The original work [13] 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 send between any neighboring GPUs. Figure 25 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 [13]). It is less efficiently 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.

5.6 Combined with PTW optimization

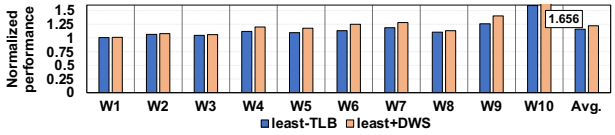

Figure 26: Combined with DWS [59].

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. [59]. They focused on mitigating PTW contention under multi-tenancy on a single GPU. Figure 26 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.

6 RELATED WORK

Address translation optimizations: There exist a substantial body of prior works focusing on optimizing address translation process in CPU apart from the ones we have discussed in Section 1 [1, 2, 10, 16, 17, 21, 40, 46, 55, 70–72, 74]. Barr et al. [11] exploited the spatial locality in physical memory allocators and used speculation for address translation to mitigate PTW overhead. Bhattacharjee et al. [19, 22] and Lustig et al. [45] proposed shared last-level TLB to accelerate parallel CPU applications by sharing translations between cores. Srikantaiah et al. [66] presented a "synergistic TLB" design that allows one TLB victim entries to be stored in other TLBs during evictions. Compared to these prior efforts, our approach employs emerging execution paradigms, multiple GPUs, and is built upon the unique execution characteristics in a multi-GPU environment. More importantly, our least-TLB hierarchy benefits both single application execution and multi-application execution where few approaches work efficiently in both execution scenarios. Meanwhile, our approach is complementary to prior PTW optimizations (e.g., [59]).

TLB optimizations on GPU: Recent works have investigated GPU TLB optimizations [8, 57, 58, 64, 65, 78, 86]. Jaleel et al. [34] proposed to use last-level cache to store cache lines and TLB entries and memorizes virtual to physical address translations in DRAM. Ausavarungnirun et al. [9] proposed to use TLB-fill token and L2-TLB bypass to avoid thrashing and improve the performance of concurrent applications at the L2-TLB level. Buruah et al. [13] exploited the TLB sharing and proposed prefetching and probing mechanisms to improve the performance of sensitive TLB applications. Tang et al. [75] proposed TLB compression and decompression mechanisms that allow more address translations to be stored in the same TLB entry. All these works focus on *single* GPU optimization. In contrast, our proposed least-TLB is the first TLB optimization that targets multi-GPU systems. Moreover, these prior works rely on fast communication within a GPU to work efficiently and deliver good TLB performance. However, the latency in multi-GPU systems is significantly higher. Our proposed least-TLB design leverages the Cuckoo filter to index instead of sending inquiries during execution, remove the need for broadcasting and receiving inquiries. This is also why our approach outperforms the state-of-the-art optimization [13], as we discussed in Section 5.4.

Software TLB optimizations: There also exist some works that proposed OS and compiler optimizations to improve the TLB performance [7, 14, 25, 43, 44]. Yan et al. [84] proposed OS support called translation ranger to reduce address translation overhead by leveraging the contiguity in access pattern. Pham et al. [54] exploited the spatial locality in page table entries and presented a multi-granular TLB organization to map multiple pages per entry to increase TLB reach. Shahar et al. [62] proposed ActivePointers, a software address translation mechanism, to support for page faults and virtual address space management for GPU programs. In our work, we enhance the IOMMU design with minimal hardware overhead. Potentially, our approach is orthogonal to the software approaches and can be combined with them to further accelerate the address translation.

7 CONCLUSION REMARKS

In this paper targeting multi-GPU systems, we comprehensively and systematically studied the address translation performance for both single application execution and multi-application execution. Our investigation reveals that there exist significant redundancy of translation in the TLB hierarchy, causing severe TLB thrashing and reducing the TLB reach. To this end, we proposed least-TLB design for multi-GPU systems. The least-TLB employs least-inclusive policy to eliminate the translation redundancy while leveraging Cuckoo filter to capture translation reuses efficiently. Experimental results show that the proposed TLB effectively improves, on average, 23.5% and 16.3% for single application execution on multi-GPUs and multi-application execution on multi-GPUs, respectively.

ACKNOWLEDGMENTS

The authors sincerely thank the shepherd for all the efforts in helping us improve the paper. The authors would also like to thank the anonymous MICRO reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2011146, a startup funding from the University of Pittsburgh, and the Pitt momentum seeding grant.

REFERENCES

[1] J. Ahn, S. Jin, and J. Huh. 2012. Revisiting hardware-assisted page walks for virtualized systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 476–487. https://doi.org/10.1109/ISCA.2012.6237041

[2] J. Ahn, S. Jin, and J. Huh. 2015. Fast Two-Level Address Translation for Virtualized Systems. *IEEE Trans. Comput.* 64, 12 (2015), 3461–3474. https://doi.org/10.1109/TC.2015.2401022

[3] AMD. 2015. AMD APP SDK OpenCL Optimization Guide.

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

[5] AMD. 2016. Graphics Core Next Architecture, Generation 3, Reference Guide.

[6] AMD Corp. 2016. I/O Virtualization Technology(IOMMU) Specification. http://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf

[7] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 27–39. https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[8] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 136–150.

[9] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 503–518. https://doi.org/10.1145/3173162.3173169

[10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[11] T. W. Barr, A. L. Cox, and S. Rixner. 2011. SpecTLB: A mechanism for speculative address translation. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 307–317.

[12] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 596–609. https://doi.org/10.1109/HPCA47549.2020.00055

[13] Trinayan Baruah, Yifan Sun, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/3410463.3414639

[14] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

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

[16] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee. 2018. Scalable Distributed Last-Level TLBs Using Low-Latency Interconnects. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 271–284. https://doi.org/10.1109/MICRO.2018.00030

[17] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

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

[19] A. Bhattacharjee, D. Lustig, and M. Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[20] A. Bhattacharjee, D. Lustig, and M. Martonosi. 2017. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00795ED1V01Y201708CAC042

[21] A. Bhattacharjee and M. Martonosi. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 29–40. https://doi.org/10.1109/PACT.2009.26

[22] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-Core Cooperative TLB for Chip Multiprocessors. *SIGPLAN Not.* 45, 3 (March 2010), 359–370. https://doi.org/10.1145/1735971.1736060

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

[24] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler. 2020. 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)*. 926–939. https://doi.org/10.1109/ISCA45697.2020.00080

[25] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[26] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *GPGPU-3: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702

[27] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104. https://doi.org/10.1145/2749460779.2750389

[28] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[29] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[30] Timothy D.R. Hartley, Umit Catalyurek, Antonio Ruiz, Francisco Igual, Rafael Mayo, and Manuel Ujaldon. 2014. Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, New York, NY, USA, 413–423. https://doi.org/10.1145/2591635.2667189

[31] Bongjoon Hyun, Youngeun Kwon, Yujeong Choi, John Kim, and Minsoo Rhu. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 1109–1124. https://doi.org/10.1145/3373376.3378494

[32] Intel. 2018. The Future of Core, Intel GPUs, 10nm, and Hybrid x86. https://www.anandtech.com/show/13699/intel-architecture-day-2018-core-future-hybrid-x86/5

[33] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. 2010. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–162. https://doi.org/10.1109/MICRO.2010.52

[34] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-Performance Address Translation by Extending TLB Reach of GPU-Accelerated Systems. *ACM Trans. Archit. Code Optim.* 16, 1, Article 6 (March 2019), 24 pages. https://doi.org/10.1145/3309710

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

[36] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer. 2015. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 343–353. https://doi.org/10.1109/HPCA.2015.7056045

[37] JEDEC. 2020. High Bandwidth Memory (HBM) DRAM 2. *Jesd235* (2020). https://www.jedec.org/standards-documents/docs/jesd235a

[38] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. Association for Computing Machinery, New York, NY, USA, 223–234. https://doi.org/10.1145/2818950.2818979

[39] Jog, Adwait. 2015. Design and Analysis of Scheduling Techniques for Throughput Processors. https://etda.libraries.psu.edu/catalog/26480

[40] G. B. Kandiraju and A. Sivasubramaniam. 2002. Going the distance for TLB prefetching: an application-driven study. In *Proceedings 29th Annual International Symposium on Computer Architecture*. 195–206. https://doi.org/10.1109/ISCA.

2002.1003578

[41] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[42] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 114–126. https://doi.org/10.1109/MICRO.2014.62

[43] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 651–664. https://doi.org/10.1145/3173162.3173198

[44] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[45] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.* 10, 1, Article 2 (April 2013), 38 pages. https://doi.org/10.1145/2445572.2445574

[46] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[47] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4, Article 69 (July 2015), 35 pages. https://doi.org/10.1145/2788396

[48] NVIDIA. 2018. DB2 Launch Datasheet Deep Learning Letter WEB. https://www.scribd.com/document/336084072/61681-DB2-Launch-Datasheet-Deep-Learning-Letter-WEB-NVidia-Deep-Learning-Box#

[49] NVIDIA. 2019. Memory Management on Modern GPU Architectures. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9727-memory-management-on-modern-gpu-architectures.pdf

[50] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border Control: Sandboxing Accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 470–481. https://doi.org/10.1145/2830772.2830819

[51] M. Parasar, A. Bhattacharjee, and T. Krishna. 2018. SEESAW: Using Superpages to Improve VIPT Caches. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 193–206.

[52] C. H. Park, T. Heo, J. Jeong, and J. Huh. 2017. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 444–456. https://doi.org/10.1145/3079856.3080217

[53] Gregory F Pfister. 2001. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O* 42, 617-632 (2001), 102.

[54] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[55] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[56] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee. 2015. 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)*. 1–12. https://doi.org/10.1145/2830772.2830773

[57] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[58] J. Power, M. D. Hill, and D. A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[59] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2021. Improving GPU Multitenancy with Page Walk Stealing. In *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*.

[60] Jihyun Ryoo, Mengran Fan, Xulong Tang, Huaipan Jiang, Meena Arunachalam, Sharada Naveen, and Mahmut T Kandemir. 2019. Architecture-Centric Bottleneck Analysis for Deep Neural Network Applications. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 205–214.

[61] Albert Segura, Jose-Maria Arnau, and Antonio González. 2019. SCU: A GPU Stream Compaction Unit for Graph Processing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 424–435. https://doi.org/10.1145/3307650.3322254

[62] S. Shahar, S. Bergman, and M. Silberstein. 2016. ActivePointers: A Case for Software Address Translation on GPUs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 596–608. https://doi.org/10.1109/ISCA.2016.58

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

[64] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 180–192. https://doi.org/10.1109/ISCA.2018.00025

[65] S. Shin, M. LeBeane, Y. Solihin, and A. Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 352–363. https://doi.org/10.1109/MICRO.2018.00036

[66] S. Srikantaiah and M. Kandemir. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 313–324. https://doi.org/10.1109/MICRO.2010.26

[67] JEDEC Standard. 2013. High Bandwidth Memory (HBM) DRAM. *Jesd235* (2013).

[68] 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. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230

[69] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581262

[70] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. 2016. Improving Bank-Level Parallelism for Irregular Applications. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[71] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2019. Computing with Near Data. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[72] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

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

[74] Xulong Tang, Mahmut Taylan Kandemir, Mustafa Karakoy, and Meena Arunachalam. 2019. Co-Optimizing Memory-Level Parallelism and Cache-Level Parallelism. In *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation*.

[75] Xulong Tang, Ziyu Zhang, Weizheng Xu, Mahmut Taylan Kandemir, Rami Melhem, and Jun Yang. 2020. Enhancing Address Translations in Throughput Processors via Compression. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*. Association for Computing Machinery, New York, NY, USA, 191–204. https://doi.org/10.1145/3410463.3414633

[76] Scott Thornton. 2021. Low cost, low latency PCIe ideal for sharing resources. Website. https://www.microcontrollertips.com/pcie-sharing-resources-faq/.

[77] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. 2008. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *2008 International Symposium on Computer Architecture*. 51–62. https://doi.org/10.1109/ISCA.2008.16

[78] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[79] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. 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*. 41–53.

[80] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. 2020. Benchmarking High Bandwidth Memory on FPGAs. *arXiv preprint arXiv:2005.04324* (2020).

[81] Jinhui Wei, Jianzhuang Lu, Qi Yu, Chen Li, and Yunping Zhao. EasyChair, 2020. Dynamic GMMU Bypass for Address Translation in Multi-GPU Systems. EasyChair Preprint no. 4179.

[82] Chenhao Xie, Fu Xin, Mingsong Chen, and Shuaiwen Leon Song. 2019. OO-VR: NUMA Friendly OBject-ORiented VR Rendering Framework for Future NUMA-Based Multi-GPU Systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3307650.3322247

[83] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 174–183. https://doi.org/10.1145/1555754.1555778

[84] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223

[85] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee. 2017. Hardware translation coherence for virtualized systems. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 430–443. https://doi.org/10.1145/3079856.3080211

[86] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. 2018. Filtering Translation Bandwidth with Virtual Caching. *SIGPLAN Not.* 53, 2 (March 2018), 113–127. https://doi.org/10.1145/3296957.3173195