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

Bingyao Li*, Jieming Yin†, Anup Holey‡, Youtao Zhang*, Jun Yang*, and Xulong Tang*
*University of Pittsburgh, Pittsburgh, Pennsylvania, USA
†Lehigh University, Bethlehem, Pennsylvania, USA
‡NVIDIA, Santa Clara, California, USA
Email: {bil35, juy9, tax6}@pitt.edu, Jieming.Yin@outlook.com, aholey@nvidia.com, zhangyt@cs.pitt.edu

2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071054

***Abstract*—Multi-GPU systems have become a popular platform to meet the ever-growing application demands. However, employing multiple GPUs does not guarantee proportional performance improvements. While prior works have extensively studied the optimizations to mitigate the non-uniform memory accesses (NUMA) overheads, the address translation process also plays an important role in shaping the overall execution performance. In this paper, we investigate the address translation process in multi-GPU systems under unified virtual memory (UVM). We specifically focus on the efficiency of page table walk and identify three major latency penalties: i) queuing for available page table walk threads, ii) memory accesses for page walk cache misses, and iii) handling page faults. Based on our observations, we propose Trans-FW, which short circuits the page table walk by leveraging substantial translation sharing and eager remote translation forwarding. Experimental results on 10 representative multi-GPU applications show that our proposed approach improves the overall performance by 53.8% on average.**

***Index Terms*—multi-GPU, page fault, page table walk**

I. Introduction

In recent years, multi-GPU systems have gained momentum to bridge the ever-increasing gap between GPU memory capacity and application demands. Commercial multi-GPU systems, including NVIDIA DGX [52] and Intel Xe [34], incorporate multiple GPUs connected through interconnections (e.g., PCIe [50] and NVLink [29]) to provide a large aggregated memory capacity through unified virtual memory (UVM). This significantly simplifies the application deployment on multi-GPU systems and has gain popularity in machine learning [65], [75], bioinformatics [21], [31], scientific computing [48], and entertainment [27], [77].

However, the promise of multi-GPUs is constrained by i) expensive address translation process and ii) non-uniform memory access (NUMA) overheads. While substantial prior works have investigated the NUMA overheads [11], [47], [49], [80], the address translation receives little attention in multi-GPUs. Specifically, translation that misses the GPU local TLB hierarchy must go through local multi-level page table walk handled by GPU MMU (GMMU). If the page is invalid in the local page table, a far fault[1] is generated and handled by the UVM driver on the CPU side which also introduce latency to the address translation process.

[1]In this paper, we use the term "far fault" and GPU "local page fault" interchangeably.

TABLE I
Comparison with prior techniques.

Techniques	**Reduce PT queuing**	**Reduce PW-cache misses**	**Reduce far fault latency**	**Write intensive**	**Multi -GPU**
TLB optimizations[38], [78] [16], [57], [59], [61], [71], [73]	✓	✓	✗	✗	✗
PW-cache design[20], [45], [58]	✗	✓	✗	✓	✗
PW-cache prefetch [46]	✗	✓	✗	✓	✗
Page replication [25], [49]	✗	✗	✓	✗	✗
Page walk scheduling[64], [66]	✓	✗	✗	✓	✗
Large page [12], [56], [60]	✓	✓	✗	✗	✗
Our approach	✓	✓	✓	✓	✓

In this paper, we identify three latency penalties in multi-GPU address translation process. First, the translation that misses the TLBs has to be queued and wait for available page table walk threads. Second, the page table walk frequently misses the MMU cache (i.e., page table walk cache). Third, there exist substantial GPU local page faults due to frequent page migration across multiple GPUs. Prior works on CPU and GPU address translation optimization are ill-suited in the context of multi-GPUs. We provide a comprehensive summary of related works in Table I. First, existing TLB optimizations (e.g., range-based TLB [78], clustering TLB [59], compression [71], least-TLB [42], and TLB probing [16]) do not optimize the page table walk latencies and cannot mitigate the frequent local page faults in multi-GPUs. Second, MMU cache prefetching is effective in accelerating page table walking. However, prefetching does not mitigate the page table walk queuing overhead nor the page fault handling overhead. Third, prior works on irregular page table walk and multi-tenancy page table walk focus on single GPU [64], [66], [67]. They are not suitable for multi-GPU executions as they do not optimize the local page faults caused by page sharing among GPUs. Fourth, page replication [25], [49] can reduce local page fault handling latency. However, it is not suitable for write-intensive applications, as a write to a page invalidates all other replications. Finally, although employing large pages improves the TLB reach, it may cause extra page faults if a large page is frequently shared among different GPUs throughout program execution, due to the increased false sharing.

Motivated by these challenges, we systematically investigate and optimize the address translation in multi-GPU systems. We observe and quantify three latency overheads in the translation process. To mitigate these overheads, we propose Trans-

FW (Translation Forwarding). The main contributions of the paper are summarized below.

- We investigate the page table walk performance in multi-GPUs. We identify three latency penalties in multi-GPU address translation process.
- We propose Trans-FW to improve the page table walk performance in multi-GPU executions. First, short circuiting is employed in GMMU by early sending local page faults to host MMU. Second, we leverage remote GPU to supply translation requests instead of waiting in the host when doing so is beneficial.
- We evaluate Trans-FW using 10 representative applications covering various data access patterns across GPUs. Experiment results show that Trans-FW achieves an average of 53.8% overall performance improvement.
- We evaluate Trans-FW with different PW-cache structures, page replication, large page, PW-cache prefetching, and TLB optimizations. The results show that Trans-FW either outperforms the existing approach by taking the multi-GPU execution characteristics into account, or works along with the existing approaches to bring additional benefits.

II. Background

A. Multi-GPU Architecture

In this paper, we target multiple discrete GPUs connected through an interconnect (e.g., PCIe and NVLink). Figure 1 illustrates the target multi-GPU architecture [6], [7], [39]. Specifically, each GPU consists of multiple Compute Units (CUs) and two levels of TLBs for address translation: i) per-CU fully associative private L1 TLB and ii) per-GPU shared L2 TLB for all CUs. Unified virtual memory [54] is employed and managed by the UVM-driver. Note that, each GPU has its own local memory and local page table. A GPU Memory Management Unit (GMMU) handles local page table walks. The GMMU comprises i) a page walk queue (*PW-queue*) to buffer the translation requests waiting for available page walk threads, ii) a page walk cache (*PW-cache*) that holds the recent translations to reduce the number of memory accesses of page table walks, and iii) multi-threaded page table walk (*PT-walk*) that handles multiple translation requests concurrently. The UVM-driver on the CPU side is responsible for coordinating all GPU far faults. The UVM driver manages a centralized page table in the host memory, which holds all valid and up-to-date address translations for all GPUs and in which GPU/CPU the physical addresses are located.

Address translation: Figure 1 also illustrates the address translation process. The memory requests generated by the same wavefront are first coalesced by the GPU memory coalescing unit. Then, the L1 data cache and the L1 TLB perform lookups in parallel in a virtually indexed physically tagged (VIPT) TLB-cache design (❶). Upon L1 TLB misses, the L1 Miss Status Holding Register (MSHR) is first checked to filter out repetitive requests, and the outstanding requests are forwarded to the L2 TLB for lookup (❷). Translations that miss in the L2 TLB and L2 MSHR are sent to the local PT-walk in the GMMU (❸). Because there is limited number of PT-walk

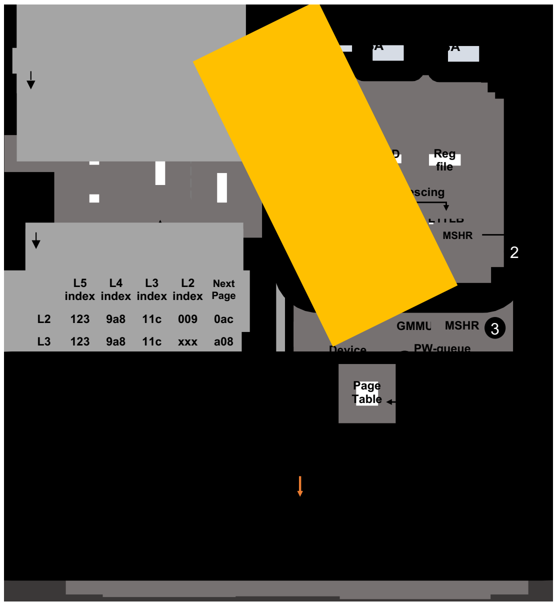

Fig. 1. Multi-GPU system-architecture.

threads, L2 TLB misses may not be served immediately. As a result, these translation requests will be stored in the PW-queue and wait for available PT-walk threads. During the page table walking, the translation is first checked in the PW-cache (❹); if it misses the PW-cache, the GPU local page table is accessed, which can be expensive and involves multiple memory accesses (❺). If the page walk fails, a far fault is propagated to the GMMU and kept in a structure called GPU Fault Buffer [6], [7]. Each time a far fault arises, the GMMU sends an alert to the UMV-driver. Upon the receipt of a far fault, the UVM driver fetches the fault information and caches them on the host side. The cached page faults are processed in batch granularity (the batch size is 256 [53]). Per batch, the UVM-driver initiates threads to perform page table walks using the centralized page table, initiates data transfer, and updates the GPU local page tables [7]. The translation request is replayed after the far fault is resolved.

B. Driver Versus Hardware Handled Far Faults

While multi-GPUs typically rely on the UVM-driver software to handle the far faults (as we discussed above), this software handled far faults creates a severe performance bottleneck which limits the scalability multi-GPUs. Recently, there has been an increasing trend for multi-GPUs to leverage hardware (e.g., host MMU/IOMMU[2] [43], [44]) to accelerate the far fault handling. Figure 1 illustrates the address translation process using hardware to handle far faults. The process within each GPU is identical to the driver handled page faults. When a far fault is generated, it is sent to the host and then handled by host MMU (①). Specifically, upon receiving a translation request, the host MMU first performs a host MMU TLB lookup. If the translation misses in the TLB, the request waits in the host MMU PW-queue for PT-walk (②). The PT-walk process in the host MMU is similar to the GPU local PT-walk, including

[2]In this paper, we use the term "host MMU" and "IOMMU" interchangeably.

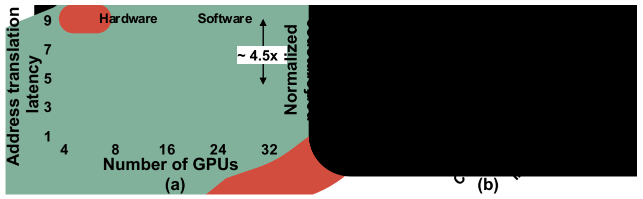

Fig. 2. (a) Normalized execution time of hardware approach versus software approach when the number of GPUs increases. (b) Performance of hardware approach normalized to software approach for all benchmarks in four GPUs.

host MMU PW-cache lookup (③), host MMU PT-walk for PW-cache misses (④), and host MMU TLB update (⑤). The translation request is replayed after the page fault is resolved, following the same process discussed above.

We quantitatively compare UVM-driver (software) handled far faults versus host MMU (hardware) handled far faults. Specifically, we carefully model the UVM-driver handled far faults (e.g., batch size and buffer size for far faults, far fault handling cost, and the number of threads for page table walk) based on prior works [6], [7], [39], [53]. We also implement hardware handled far faults as discussed above. Figure 2(a) shows the scalability of software versus hardware when the GPU count increases. All the results are normalized to the hardware approach with four GPUs. As one can observe, the address translation overhead of software is 4.5× higher than that of hardware in 32 GPUs. There is an increasing gap between software and hardware implementations, making the software approach one of the main scalability constraints in multi-GPU systems. Figure 2(b) shows the performance of hardware approach normalized to software approach in four GPUs using our targeted applications (details discussed shortly in Table III). We observe that the hardware approach outperforms the software approach by up to 56.3% and 28.4% on average.

Thus, unless specified otherwise, we adopt the hardware handled far faults as the **baseline** in our paper for a fair comparison. We identify three major latencies in the baseline address translation: i) waiting time for available PT-walk threads in the PW-queue (❸ and ②). ii) additionally memory accesses after PW-cache misses (❺ and ④) and iii) handling far faults caused by page sharing among multiple GPUs (①, ②, ③, ④, and replay). We quantitatively investigate these latencies in Section III-B.

Page walk cache: In this paper, we adopt the Unified Translation Cache (UTC) [22] in both the baseline and our approach. Specifically, the cache entry in UTC is maintained separately for each page level tagged with indices in the virtual address (i.e., an L5 entry is tagged with the L5 index, an L4 entry is tagged with the L5 and L4 indices, and so forth). Therefore, a single lookup can provide the longest matching prefix, avoiding nested PW-cache lookups. In UTC structure, entries from different levels of the page table are mixed in a single cache. Figure 1 also shows an example of the UTC structure. After the PT-walk for translating the virtual address (123, 9a8, 11c, 009, 1b8), the contents in PW-cache are (123/9a8/11c/009, 0ac) as L2 entry, (123/9a8/11c, a08) as L3 entry, (123/9a8, 116) as L4 entry, and (123, 8be) as L5 entry. If a subsequent query tries to translate the virtual address (123, 9a8, 11c, 026, 00b), the PT-walk first checks the PW-cache. Three matching entries can be found, i.e., (123, 8be) in L5, (123/9a8, 116) in L4, and (123/9a8/11c, a08) in L3. Then the L3 entry is used because this tag matches the longest prefix of the virtual address. Note that, all the contents in the **index** fields store bits from virtual address, and the **Next Page** field stores physical address. For the same virtual address, the PW-cache index content is the same, and only the Next Page field is different because they correspond to different GPUs/CPU.

III. Motivational Study

A. Baseline Configuration and Applications

We conduct our characterization and later evaluate our proposed Trans-FW using the MGPUSim [69]. MGPUSim supports multi-GPU simulation and is validated against industrial multi-GPU systems [9]. To model unified virtual memory and the full address translation process, we substantially modified and extended MGPUSim by adding i) per-GPU GMMU module with GPU local page tables, local PW-queue, and local PW-cache, and ii) host MMU module with a host TLB, host MMU page table, host MMU PW-cache, and host MMU PW-queue.

TABLE II
Baseline multi-GPU configuration.

Module	Configuration
CU	1.0 GHz, 64 per GPU
L1 Vector Cache	16 KB, 4-way
L1 Inst Cache	32 KB, 4-way
L1 Scalar Cache	16 KB, 4-way
L2 Cache	256 KB, 16-way
DRAM	4 GB
L1 TLB	32 entries, 32-way, 1-cycle lookup latency, CU private, LRU replacement policy
L2 TLB	512 entries, 16-way, 10-cycle lookup latency, CUs shared, LRU replacement policy
Host MMU TLB	2048 entries, 64-way, GPUs shared, LRU replacement policy
Page table walk	Host MMU 16 shared page table walker, GMMU 8 shared page table walker [64], [66], [74], 100-cycle latency per level [32]
Page walk cache	128 entries shared across page table walker [64]
Page walk queue	64 entries
CPU-GPU interconnection	PCIe, 150-cycle latency [32]

TABLE III
List of applications.

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

Baseline GPU configuration: In this paper, we target a 4-GPU system where each GPU has its own page table stored in its device memory [43], [44], [76]. The detailed baseline configurations are listed in Table II. Note that, our approach is also applicable to different GPU counts and we provide a sensitivity study with 8 and 16 GPUs in Section V-B. We employ a five-level nested page table organization, thereby a four-level PW-cache [33]. We also evaluate four-level page table in Section V-B. We use UTC PW-cache organization. The total size of the PW-cache is typically in the range of 64 to 128 [14], [45], [64] under a four-level page table. We employ 128-entry PW-cache in our five-level page table in GMMU and host MMU. The CTA policy in the baseline and our approach is as follows. The CTA scheduler first schedules the CTAs across CUs within a GPU in a round-robin fashion, and then moves to the next GPU only when the GPU has no available resources. That is, the CTA is scheduled greedily across GPUs. This scheduling captures the inter-CTA locality within a GPU and also maintains computing balancing across CUs.

Applications: We use 10 applications from Hetero-Mark [68], AMDAPPSDK [8], SHOC [24], and DNN Mark [26] benchmark suites. The details of the applications are listed in Table III. We use their multi-GPU implementation from [69], which is also used by prior works [15], [42], [70]. The applications cover a wide range of data access/sharing patterns in multi-GPU environment. We classify these applications into four categories based on their memory access patterns: random (`PR`), partition (`AES`), adjacent (`ST, FIR, SC, Conv2d, KM`), and scatter-gather (`MT, MM, Im2col`).

B. GPU Page Walk Characterization

Fig. 3. Latency breakdown of GPU L2 TLB misses.

Recall the three latencies discussed in Section II-B, Figure 3 shows the latency breakdown of GPU L2 TLB misses. One can make the following observations. First, handling local page faults causes significant overhead and accounts for 86.1% of the L2 TLB miss latency. The local page faults latency can be further breakdown into i) waiting in the shared host MMU PW-queue, ii) missing the host MMU PW-cache, iii) migrating page to local memory, and iv) CPU-GPU interconnection and request replayed when page fault is resolved. Second, 25.0% of the latency is caused by requests waiting for the available page table walk thread in the PW-queue. The PW-queue queuing latency in shared host MMU is generally longer than that in the GMMU for most applications. Specifically, the average queuing latencies are 4.1% and 20.9% for the GMMU PW-queue and the host MMU PW-queue, respectively. This is because the host MMU handles page faults generated from all GPUs, encountering severer contention than local GPUs. Finally, an average of 9.0% and 9.3% latency is caused by GMMU and host MMU PW-cache miss, respectively.

Room for improvement: We study the performance gains when we i) adopt infinite PW-cache in both GPU and host MMU, ii) employ infinite page table walking threads in both

Fig. 4. Performance improvements when latency sources are resolved.

GPU and host MMU, and iii) eliminate all GPU local page faults. The normalized execution performance is shown in Figure 4. In the results, we separate the influence of each latency by applying only one of the above three impractical optimizations at a time. For example, when we adopt infinite PW-cache, the other two configurations remain the same as in the baseline. We next discuss the implication from each individual optimization in detail.

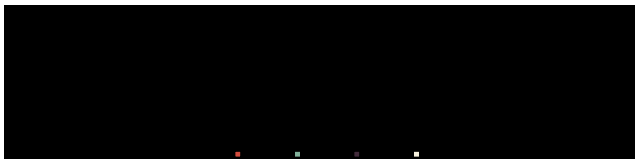
Fig. 5. GMMU PW-cache hit rate in the baseline.

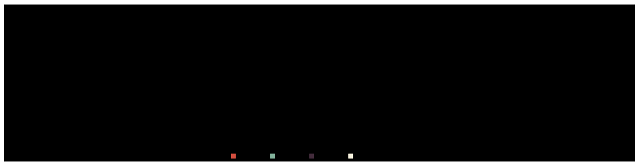
Fig. 6. Host MMU PW-cache hit rate in the baseline.

Implication of infinite PW-cache: An infinite-sized PW-cache only has cold misses. We employ infinite-sized PW-cache in both GMMU and host MMU. The first bar of each application in Figure 4 shows the performance improvement of an infinite PW-cache. Overall, the infinite PW-cache achieves up to 29.6% speedup, with an average performance improvement of 16.2%. To better understand the improvement, we report the hit rates for GMMU and host MMU PW-caches in Figure 5 and Figure 6, respectively. For GMMU PW-cache, 56.6% of page table walks can get the translation within 1 to 2 memory accesses and the rest of the translations must go through 3 to 4 memory accesses (i.e., walking the page table). For host MMU PW-cache, we observe 47.5% hit rates in the higher levels (e.g., L4) and 50.9% hit rates in the lower levels (e.g., L3 and L2). Therefore, multiple memory accesses are still required for address translation in the host MMU. Since the memory accesses are long latency operations, poor PW-cache hit rate can degrade the performance.

Implication of infinite PT-walk threads: With infinite page table walking threads in both GMMU and host MMU, all TLB misses are served immediately without waiting in the PW-queues. Referring to the second set of bars in Figure 4, we observe an average of 42.6% performance improvement over the baseline. The performance gain is higher for applications (e.g., `PR`, `SC`, and `MT`) whose waiting latency occupies higher percentages of the L2 TLB miss latency.

Implication of eliminating GPU page faults: Eliminating all GPU local page faults ensures that all translations are found in the GPU local page tables, leading to an average of 2.2× performance improvement (the fourth bar of each application in Figure 4). We quantify the *page-faults-per-kilo-instructions (PFPKI)* and show the PFPKI of each application in Table III. We observe that applications with substantial page sharing among multiple GPUs (detailed study in Section III-C) have higher PFPKI values (e.g., `MT`, `PR`, and `SC`). This is because page sharing causes frequent page migration among GPUs, and hence subsequent page accesses to the migrated page will generate GPU local page faults. In contrast, applications with less page sharing across the GPUs (i.e., each GPU works on its own local data partitions) have lower PFPKI, such as `AES` and `FIR`. We also observe that applications with higher PFPKI achieve relatively high performance improvement (e.g., `MT` and `SC`), whereas lower PFPKI show less performance improvement (e.g., `AES` and `FIR`). However, `ST` has a high PFPKI but low performance improvement. This is because the contention in the host MMU page table walk is less severe in `ST` execution. In contrast, `Conv2d` has a low PFPKI but a high performance improvement. This is because a large number of pending requests are coalesced to the same page fault in L2 MSHR. Reducing the time for handling a page fault can significantly benefit the whole execution.

We show the performance improvement where we eliminate the data page migration latency while preserving the address translation latency (third bar in Figure 4). The results show an average of 63.9% performance improvement over baseline. Comparing the results with the above study, which eliminates the local page faults (2.2× performance improvement), one can observe that the address translation plays an important role in improving the performance, and there exists a large optimization potential.

C. Page Sharing Characterization

Fig. 7. Percentage of page sharing.

Observation 1: *There exist substantial page sharing among multiple GPUs.* Figure 7 shows the page sharing among multiple GPUs during the execution of each application. Specifically, we define page sharing ratio as the percentage of shared page accesses divided by the total number of page accesses during the execution of the application. As one can observe, a significant fraction of pages is shared by multiple GPUs. For example, in `MM`, `PR`, `KM` and `SC`, almost all pages are shared by all four GPUs. In `MT` and `Conv2d`, about half of the pages are shared between two or four GPUs. Such intensive page sharing also brings significant address translation sharing among the GPUs.

Observation 2: *Local PW-cache misses and page faults can be resolved by other GPUs.* We investigate whether a local

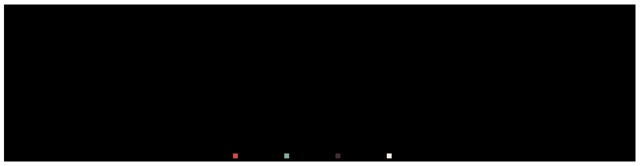

Fig. 8. Remote PW-cache hit rate

page fault can find the translation in other GPUs' PW-caches. The intuition behind is to leverage the translation reuses that stem from page sharing among GPUs to reduce the number of PW-cache misses and the latency of handling local page faults. To capture this opportunity, we define *remote* PW-cache hit as the percentage of local page faults that hits remote GPU's PW-cache. We observe 88.2% remote hit rate (all levels combined) in Figure 8. It is also important to note that, an average of 45.2% can hit in lower levels of PW-cache (e.g., L3 and L2), indicating that only 1-2 memory accesses are needed to get the translation from the remote GPU.

IV. Our Approach

A. High Level Overview

In this paper, we propose Trans-FW that leverages the substantial translation sharing among GPUs and employs the remote GPU to serve the translation requests when doing so is beneficial. There are three major challenges to implementing an effective and efficient remote forwarding scheme. First, the remote GPU supplies the translation request only if it holds the valid page. Therefore, it is important to determine which GPU has the valid page. Second, accessing remote page tables may take longer than accessing the host page tables due to network congestion and remote PT-walk contention (e.g., PW-queue waiting and PW-cache misses). Thus, it is important to dynamically detect the beneficial scenarios and provide flexible and efficient hardware support for guiding and handling translation requests remotely. Finally, the proposed scheme should have minimal hardware overhead and is light-weight compared to enlarging the TLB capacity. To this end, we propose Trans-FW. Figure 9 shows the high-level overview of our design.

B. Short Circuiting in GMMU

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

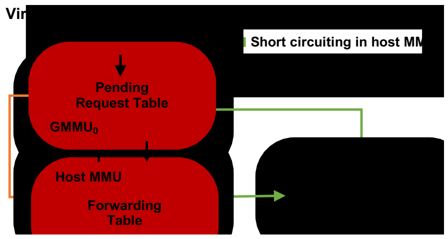

Fig. 9. High level overview of Trans-FW.

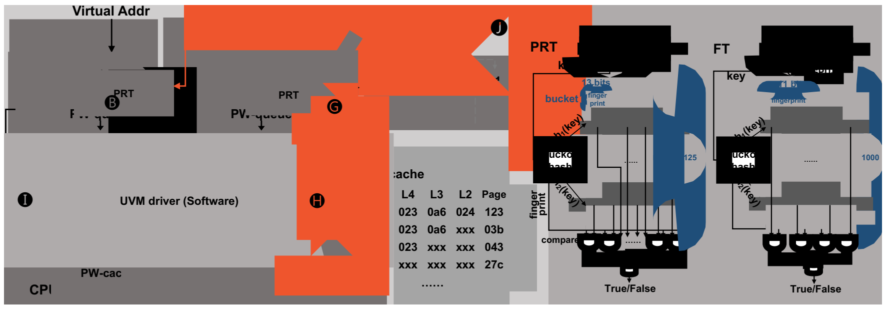

Fig. 10. Translation lookup in Trans-FW, and details of PRT and FT.

host MMU because the number of GPU L2 TLB misses is significantly larger than the GPU local page faults. Therefore, a naive implementation of short-circuiting the GMMU may cause CPU-GPU interconnection congestion and PT-walk contention of the host MMU. To mitigate the contention, we propose a Pending Request Table (PRT) in each GMMU as a filter to reduce the number of requests sent to the host MMU. We next discuss the design of PRT.

Pending request table: The PRT tracks the translations of all pages that reside in the GPU's local memory. A Cuckoo filter [28] is implemented in the PRT, which is a hardware-efficient structure that supports fast set membership testing. Figure 10 shows the microarchitectural details of PRT (green box). Specifically, the PRT consists of 125 buckets and each bucket comprises four fingerprints. Each inserted item (i.e., VPN) is converted into a fingerprint using the cuckoo hash functions (i.e., h_1 and h_2, MetroHash hash function [35] is employed in our approach.). A fingerprint is formed by using the virtual page number. The Cuckoo filter provides efficient insertion and deletion operations. It inserts the fingerprint of an entry into one of the two alternative buckets indicated by the two hash functions. If neither bucket has space, the fingerprint selects one of the candidate buckets, kicks out the existing fingerprint, and re-inserts this victim fingerprint into its own alternate location. When looking up an item, the Cuckoo filter first calculates the item's fingerprint and the two candidate buckets. If any existing fingerprint in either bucket matches the request, the cuckoo filter returns true. Otherwise, it returns false. In our current design, the PRT comprises eight comparators so it can check all the fingerprint candidates from the two buckets in parallel. Deletion is supported by removing one copy of matched fingerprint from any candidate bucket. Note that, when two identical fingerprints are stored in the two checked buckets, a random one is selected and deleted, which causes false positive cases. Note that, the PRT needs to be updated when a page is migrated. Specifically, when a page is migrated away from the GPU, the virtual page number is used to locate the fingerprint. Then, this fingerprint is removed from the PRT. When a new page is migrated to the GPU, a new fingerprint is formed and inserted into the PRT. This update progress is off the execution critical path and can overlap with and be hidden by GPU execution.

Lookup procedure: When a translation request misses in the GPU local L2 TLB, the PRT is checked first. If the request misses the PRT, which indicates that the requested translation is definitely invalid in the local GPU page table and the page is not presented locally (since the cuckoo filter has no false negative cases), then the request is early forwarded to the host MMU without a GMMU PT-walk. If the request hits the PRT, which indicates a high potential of finding a translation locally, then the request is sent to the GMMU PT-walk for local page table lookup. Note that, it can happen that the Cuckoo filter provides a false prediction and the GPU doesn't hold the translation. A local page fault will be generated after the GMMU PT-walk, which is the same as the baseline execution. However, considering a low false positive in our configuration, this scenario rarely happens and its additional latency has little impact on the overall performance.

C. Short Circuiting in Host MMU

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

How to borrow: To borrow PT-walk from a remote GPU, it is important to first determine which GPU has the valid page. If a remote GPU does not hold the valid page, there is no performance gain since the remote GPU will also generate a page fault on the request. Therefore, we implement a Forwarding Table (FT) in the host MMU to indicate which GPU has the valid page. The FT leverages a similar Cuckoo filter design as in PRT to guide the forwarding of the requests to the remote GPUs. Figure 10 also shows the microarchitectural details of FT (grey box). Specifically, the FT has 1,000 buckets. The key used by the two cuckoo hash functions is a concatenation of VPN and GPU_{id}. Since each bucket only has two fingerprints, we implement four comparators to perform parallel item lookups. A fingerprint is formed by concatenating the virtual page number and the owner GPU ID (i.e., the GPU who has the valid page) when the fingerprint is inserted into the FT. The FT is updated when a page is migrated. First, the page number and prior owner GPU ID are used to locate the fingerprint. Then, the old fingerprint is deleted from the FT and replaced with the new one, which is the concatenation of the page number and the new owner GPU ID. Note that, the FT supports four parallel GPU ID lookups. That is, four different GPU IDs are searched using one of two hash functions. If the desired fingerprint is found, it only takes one cycle. Otherwise, the four GPU IDs are searched using the other hash function. Note also that, identical fingerprints may store in the checked buckets, a random one is selected and deleted, the desired fingerprint may not be deleted. A new fingerprint is then inserted with the same page number but different owner GPU ID, which will result in two or more different owner GPU IDs for the same page number storing in the FT. As a result, when a request looks up the FT, it will return multiple owner GPU IDs. In the case, our design chooses any one and forwards the request to that GPU.

When to borrow: When a request arrives at the host MMU, the host MMU TLB and FT are searched in parallel. If the request hits in the host MMU TLB, the translation is directly returned to the requesting GPU. If the request misses in the host MMU TLB but hit in the FT, we will check the number of requests queued in the host MMU PW-queue and use it as the indicator for the host MMU PT-walk contention. Depending on the contention in host MMU PT-walk, two scenarios may happen. First, we observe when the number of queued requests is less than half of the PT-walk threads (we call it as the forwarding threshold)[3], a host MMU lookup is usually faster than a remote lookup, considering the remote GMMU contention and the network latency. In this scenario, we only rely on the host MMU to perform PT-walk without involving any remote GPUs. Second, if the number of queued requests is more than half of the number of PT-walk threads, the request is inserted into the host PW-queue but also forwarded to a remote GPU according to the FT in order to short circuit the PT-walk in the host MMU. When the desired translation is found in a remote GPU, the remote GPU will directly send the translation to the requesting GPU. The remote GPU will also notify the host MMU of success or failure after it performs the lookup. If the remote lookup succeeds, we check the host MMU PW-queue, if the request still waits in the PW-queue, the request is removed from the PW-queue to reduce the PT-walk contention. On the other hand, if the remote lookup fails due to the false positive in FT, the request will be discarded since the pending request has been sent to the host MMU PT-walk. Note that, it can happen that both host MMU and successful remote lookup will send the desired translation to the requesting GPU. However, the Trans-FW ensures that only the early returned translation will be used, and the latter one will be discarded. Note also that, in practice, we do not see many of these cases happen. This is because the queuing latency of the host PW-queue is generally large. In most cases, the host MMU will receive the forwarding GPU message and the request is removed from the host MMU PW-queue.

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

D. Putting All Together

Figure 10 illustrates the lookup procedure in Trans-FW. Specifically, when a translation request misses in GPU local L2 TLB, the request is checked in the Pending Request Table (Ⓐ). If the request hits in the PRT, GMMU PT-walk is performed (Ⓑ). Otherwise, the request is sent to the UVM driver and handled by the host MMU (Ⓒ). When the request arrives at the host MMU, the host MMU TLB (Ⓓ) and Forwarding Table (Ⓔ) are searched in parallel. If the request hits in the host MMU TLB, the translation is returned to the requesting GPU. If the request misses in the host MMU TLB, PT-walk in host MMU is performed (Ⓕ). Depending on the number of queued requests in the host MMU, the request is forwarded to the corresponding remote GPU (Ⓖ) indicated by the FT. After remote GPU performs the lookup, the translation returns to the host MMU (Ⓗ) as well as to the requesting GPU(Ⓙ). The requesting GPU uses the translation either returned from the local GMMU, host MMU PT-walks (Ⓘ), or received from the remote GPU (Ⓙ) accordingly.

E. Hardware Overhead

The sizes of PRT and FT are determined by the fingerprint size (f) in Cuckoo filter and the number of fingerprints. The fingerprint size is determined by the bucket size (b) and desired false positive rate (ε). A small fingerprint size minimizes the hardware overheads but increases the false positive rate, which reduces the performance gains. On the other hand, a large fingerprint size is beneficial to overall performance but suffers from hardware overheads. We choose 0.1% and 0.2% false positive rates for PRT and FT to strike a balance between performance and hardware overheads. Taking false positive rate of 0.2% as an example, we show the fingerprint size calculation below. According to [28], when the targeted false positive rate equals to 0.2%, two entries per bucket minimizes space. Therefore, the bucket size is 2 in this case. To retain the target false positive rate ε, the minimal fingerprint size required is approximate: $f \geq log2(1/\varepsilon) + log2(2b)$. Therefore, with a 2-bucket size, a false positive rate of 0.2% would require $\sim$ 9 + 2 = 11 bits for a fingerprint.

In our design, the Cuckoo filter in Forwarding Table (FT) has a total of 2000 fingerprints. We mask the last three bits of the virtual address so that eight pages can map to the same fingerprint. The bucket size is two. Therefore, each fingerprint is 11 bits with 0.2% predefined false positive probability. The total size of the FT is $2000 \times 11/8/1024 = 2.68$KB. The Pending Request Table (PRT) is designed with 500 fingerprints for each GMMU. Eight pages map to the same fingerprint. The bucket size is four. Therefore, each fingerprint is 13 bits with 0.1% predefined false positive probability. The total size of the PRT is $500 \times 13/8/1024 = 0.79$KB. We use CACTI [72] to estimate the areas and the results show that PRT and FT are 1.01% and 1.95% compared to the areas of GPU L2 TLB and host

MMU TLB, respectively. Note that, using the same area for large host MMU TLB cannot achieve equivalent performance improvement (we show the impact of host MMU TLB size in Section V-B).

V. Evaluation

In this section, we evaluate Trans-FW using MGPUsim [69]. The detailed simulation configurations and the applications are the same as in our characterization (listed in Table II and Table III). We use application end-to-end execution time to compute the normalized performance.

A. Overall Performance

Fig. 11. Normalized performance of Trans-FW.

Figure 11 shows the overall performance improvements of Trans-FW normalized to the baseline. Trans-FW achieves an average of 53.8% performance improvement across 10 applications. The performance improvements are significant for those applications with high PFPKI (shown in Table III). For example, `MT` achieves over 2× over the baseline. In contrast, the performance improvements of `FIR` and `AES` are marginal. This is because these two applications are compute-intensive and the page fault latency can be hidden by the light-weight context switching in GPUs. Also, these two applications have few page sharing and less local page faults. As such, these two applications are insensitive to page fault latencies.

Fig. 12. The reduced percentage of each latency component in Figure 3.

Fig. 13. PW-cache hit rates at levels L2 and L3 in baseline and Trans-FW.

To understand the reasons behind the performance improvements, we plot the percentage reduction of each latency component quantified in Figure 3. One can make the following observations. First, Trans-FW significantly reduces the PW-queue waiting time by an average of 95.8% and 79.8% in GMMU and host MMU, respectively. The reductions in PW-queue waiting time directly translate to performance improvements. Second, Trans-FW reduces the latency of address translation parts of handling local page faults by 43.4%. Specifically, for 8 applications (except `AES` and `FIR`) that have substantial page sharing, our approach reduces the latency up to 53.3%. This is because page sharing among GPUs provides opportunities for local faults to be served by remote GPUs, thereby reducing page fault latency. Third, Trans-FW also reduces the latency caused by PW-cache misses. We also show average hit rates at L2 and L3 levels of both GMMU PW-cache and host MMU PW-cache in Figure 13. Note that, the host MMU PW-cache hits comprise the remote hit enabled by Trans-FW. As shown in Figure 13, compared to the baseline host MMU PW-cache L2 and L3 hit rates in Figure 6, Trans-FW achieves a higher hit rate. This is because the remote GPU has recently accessed the requested page, and a longer translation prefix is available in the remote PW-cache. However, the hit rate of GMMU PW-cache in Trans-FW is slighter lower than the baseline. This is because when serving remote requests, the newly inserted entry causes the local PW-cache thrashing. Overall, Trans-FW improves the hit rates of PW-cache and achieves latency reduction caused by PW-cache misses.

Fig. 14. Percentage of replicated PT-walk to all host MMU PT-walk.

Recall that, when the number of queued requests in the host MMU PW-queue reaches a threshold, requests waiting in the host MMU PW-queue will also be forwarded to the remote GPU. This generates replicated PT-walk requests in the host MMU and the remote GMMU. Figure 14 shows the percentage of replicated PT-walk requests introduced by our approach to all host MMU PT-walk requests. One can observe that, our approach generates an average of 22.6% of replicated PT-walk requests to host MMU and remote GMMU. However, we want to emphasize that these additional PT-walk requests do *not* necessarily increase the overall total number of PT-walk memory accesses. The reasons are two-fold. First, short-circuiting in the host MMU allows the PT-walk requests to benefit the remote GPU's PW-cache. In our results, 65% of those replicated requests (i.e., 22.6%) in host MMU are eliminated before they incur any memory accesses because the translation is resolved by the remote GPU. Second, although those additional PT-walk requests cause 13.4% of additional PT-walk memory accesses in the GMMU (percentage calculated by the number of additional PT-walk memory accesses dividing the total GMMU PT-walk memory accesses), we short-circuit the local GMMU PT-walk for potential local page faults, which reduces an average of 49.6% of the total GMMU PT-walk memory accesses. Therefore, the total number of PT-walk memory accesses in GMMU is reduced by 36.2% (49.6% minus 13.4%) compared to the baseline. In summary, our approach reduces the total PT-walk memory accesses in both GMMU and host MMU.

B. Sensitive Study

Forwarding threshold: Recall that, in our configuration of Trans-FW, the forwarding threshold is set to half of the PT-walk

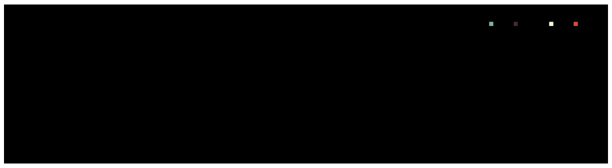

Fig. 15. Remote forwarding threshold.

threads (Section IV-C). That is, when the number of queued requests exceeds half of the number of PT-walk threads, the request is forwarded to the remote GPU. In this study, we evaluate the performance impact under different forwarding thresholds in Figure 15. First, when the forwarding threshold is set to 0 (i.e., the request is forwarded right away when there are no immediate available PT-walk threads), the average performance improvement is 51.4% over the baseline, which is 2.4% lower than setting the forwarding threshold equal to 0.5. This is because it introduces more contention in the remote GPU PT-walk. Second, the performance improvement decreases as the threshold increases. The results show an average performance improvement of 41.0% and 16.7% when thresholds are set to 1 and 2. The main reason behind this is the increasing queuing time of host PW-queue.

Fig. 16. The performance of Trans-FW with different sizes of PRT and FT normalized to the baseline execution. The (x,y) on the legend indicates (size of PRT, size of FT).

FT and PRT size: We evaluate the performance of Trans-FW under different FT and PRT sizes in Figure 16. When PRT is 250 entries (i.e., fingerprints) and FT is 1000 entries (i.e., fingerprints), the average performance improvement is 46.3% on average. This is 7.5% lower than the default sizes (i.e., 500 in PRT and 2000 in FT). This is because more pages are mapping into the same fingerprints with smaller table sizes, which causes a higher false positive rate and results in more additional latency. When PRT and FT sizes are increased to 1000 and 4000 fingerprints, the average performance improvement is slightly higher than the default size. Considering the hardware overhead, we use 500 and 2000 as our configuration.

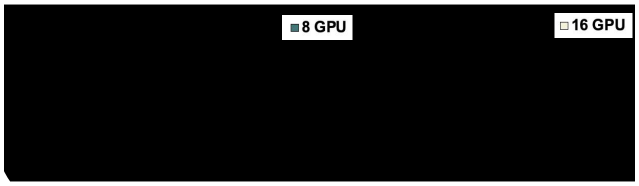

Fig. 17. Performance of Trans-FW with 8 and 16 GPUs. Results are normalized to 8-GPU and 16-GPU baseline, respectively.

Number of GPUs: We evaluate Trans-FW in 8-GPU and 16-GPU systems to show its scalability. Note that, for a fair comparison, when we increase the number of GPUs, we do not increase the application input size. Figure 17 shows the performance of Trans-FW with 8 GPUs and 16 GPUs normalized to the baseline with 8 GPUs and 16 GPUs. We observe average improvements of 50.5% and 46.1% in 8 GPUs and 16 GPUs, respectively. The performance improvement is reduced as the number of GPUs increases. This is because, with more GPUs, more page faults will be generated and sent to host MMU due to the more frequent page sharing across GPUs, causing severer contention in the host MMU. Accordingly, the page faults handling time increases in the host MMU.

Fig. 18. The performance of baseline and Trans-FW with different numbers of PT-walk threads normalized to the baseline execution. The (x,y) on the x-axis indicates (# of PT-walk threads in GMMU, # of PT-walk threads in host MMU).

Page table walk (PT-walk) threads: We evaluate Trans-FW under different numbers of PT-walk threads. Figure 18 shows the performance results. All the results are normalized to baseline execution with four-threaded PT-walk in GMMU and eight-threaded PT-walk in host MMU. One can make the following observations. First, Trans-FW outperforms the baseline in all configurations, achieving an average performance improvement of 56.8%. Second, increasing the number of PT-walk threads improves the performances in both the baseline and Trans-FW. This is because the PW-queue waiting time is effectively reduced, and the baseline benefits more from the increasing PT-walk. Finally, Trans-FW with fewer PT-walk threads achieves higher performance than baseline with more PT-walk threads. For example, Trans-FW (second bar) in (8,16) configuration outperforms the baseline (first bar) in (64,128) configuration by 13.4%. This is because Trans-FW not only reduces the PW-queuing latency, but also reduces the PW-cache miss penalty and the unnecessary PT-walk latency caused by local page faults. This also indicates that simply employing a large number of PT-walk threads in the baseline cannot achieve comparable performance with our approach.

Fig. 19. Performance of Trans-FW with 4-level page table normalized to baseline with 4-level page table.

4-level page table: We evaluate the performance when using a 4-level page table in both baseline and Trans-FW (shown in Figure 19). The results indicate that Trans-FW yields an average of 49.5% performance improvement over the baseline.

Host MMU configuration: Next, we evaluate Trans-FW using different MMU configurations. First, we employ a larger host MMU TLB size with 4096 entries (64-way and 64 sets) compared to our main results using 2048 entries in Table II with 64-way and 32 sets. Figure 20 (a) shows that Trans-FW achieves

Fig. 20. Performance of Trans-FW with different host MMU configurations normalized to baseline with corresponding configurations.

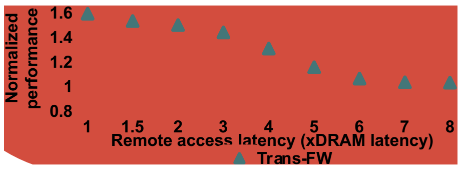

Fig. 21. Remote access latency.

54.2% performance improvement over the baseline using 4096 entry TLB, indicating that our approach is scalable with large TLBs. We observe the performance improvement is similar to 2048 entry TLB capacity. This is because a large TLB does not help due to the TLB shootdown caused by page migration. Therefore, the size of host MMU TLB has less impact on our approach, and simply increasing host MMU TLB size cannot bring comparable performance as ours in multi-GPU system. Second, we evaluate Trans-FW under different host MMU PW-cache sizes. Figure 20 (b) and (c) show the performance of Trans-FW with 256-entry and 512-entry host MMU PW-cache normalized to baseline execution with 256-entry and 512-entry host MMU PW-cache, respectively. We observe the average performance improvements of Trans-FW are 48.9% and 43.1% over the corresponding baselines. This is because a larger host MMU PW-cache does not help much in reducing long host PW-queue queuing latency. Therefore, Trans-FW still achieves significant performance improvement compared to the baseline.

Remote access latency: We evaluate Trans-FW under different remote GPU access latencies to show the cross-over point when accessing a remote GPU compared against invoking the PT-walk in the host MMU. Figure 21 shows the performance of Trans-FW when varying the GPU interconnection latency to multiple times of GPU memory latency normalized to baseline execution. One can observe that, the performance improvement of Trans-FW decreases when the remote access latency increases. In particular, when the remote latency is beyond 8× of the GPU local memory latency, the performance of accessing the remote GPU is almost the same as accessing the page table in the host MMU. This is because, at this point, the remote access latency is comparable to the waiting latency of PT-walk. Note that, the remote latency is typically 100-150 cycles as employed by prior works [2], [30], [32], we use 150 cycles in the main evaluation.

C. Different PW-cache Structure

Fig. 22. Performance of Trans-FW with STC structure normalized to baseline with STC.

In both the baseline and our approach, we adopt the UTC PW-cache organization. However, our approach is not tied to a specific PW-cache organization. To verify, we evaluate the performance impact when employing a different PW-cache design. Specifically, we implement Split Translation Cache (STC) [22]. In STC, entries from different levels of page table are stored in separate caches. The configuration of STC is as follows: 16 entries for L5, 16 entries for L4, 32 entries for L3, and 64 entries for L2 [22]. The results indicate an average of 54.0% performance improvement of our approach with STC normalized to the baseline STC, which is slightly higher than the UTC PW-cache design used in our main results. This is because each level of cache does not compete with each other in STC, and more low-level PW-caches can be saved, thereby reducing the number of memory accesses during the page table walk. Overall, Trans-FW works with different PW-caches and is able to yield significant performance improvements.

D. Page Replication

Fig. 23. Performance of Trans-FW with page replication normalized to baseline with page replication.

The unified virtual memory in modern GPU systems supports read-replication, in which a page is allowed to be accessed and replicated in multiple GPUs' physical memory with different physical addresses [54]. When a local page fault happens, a read-only page is replicated in the faulting GPU physical memory. To ensure consistency, when a GPU performs a write operation, it invalidates all the page replications in other GPUs, similar to cache coherence protocol. An ESI (Exclusive, Shared, and Invalid) memory coherency protocol is employed. When a write to a read-replicated page (S state) occurs, a protection fault is triggered. The GMMU dispatches an interrupt to the CPU to invoke the fault handler. The handler then invalidates the page from all other owners and shoots down all stale TLB entries in both the host MMU and GMMU except for the GPU that executes the write operation. Once the GPU receives the completion message of fault handling, it will re-execute the write. We evaluate Trans-FW under read-replication. Figure 23 illustrates the performance of our approach with page replication normalized to baseline execution with page replication. We observe an average of 28.4% performance improvement brought by Trans-FW. Comparing the results with the performance in Figure 11, the improvement is less. This is because that read replication effectively reduces the number of local page faults and potentially improves the PW-cache hit rates. However, for some applications (e.g., `MT`, `Conv2d`, and `Im2col`), our approach still significantly outperforms the baseline read-replication. To understand the reason behind,

Fig. 24. Percentage of reads and writes to all shared pages.

we quantify the read operations and writing operations to shared pages across GPUs in Figure 24. We observe that read-replication has marginal improvements for write-intensive applications due to the frequent write-invalidation (e.g., `MT`, `Conv2d`, and `Im2col`). In contrast, our approach is able to improve applications exhibiting both write-intensive and read-intensive page sharing.

E. Remote Mapping

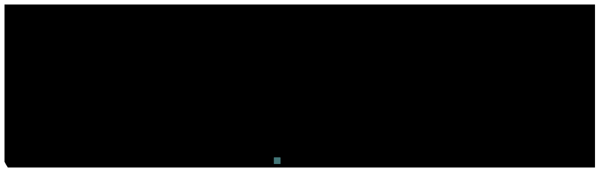
Fig. 25. Performance of Trans-FW with remote mapping normalized to baseline with remote mapping.

Our discussion so far leverages the on-touch migration which is the default scheme in modern GPUs [51], [54]. UVM also supports another page migration scheme: remote mapping. That is, a GPU can establish a direct mapping and access the remote GPU's physical memory without actually migrating the page. Instead of migrating a page, the first access to a page that is not locally available will generate a page fault and then create a corresponding page table entry that points to the page in remote GPU memory. Pages are only migrated after reaching accessing thresholds. When a page migration happens, the translations that map to that page from different GPUs have to be invalidated to ensure translation coherence across GPUs. We evaluate Trans-FW with remote mapping. Specifically, we leverage the access counter as in recent NVIDIA GPUs [53], [54] to determine the page migration. Figure 25 shows the performance of our approach with remote mapping normalized to baseline execution with remote mapping. The results indicate that Trans-FW achieves an average of 39.3% performance improvement. This demonstrates Trans-FW works with remote mapping and improves the performance when using remote mapping. The improvement is less compared to the results in Figure 11. The main reason is that remote mapping helps reduce the number of local page faults and the amount of page thrashing, especially for applications with substantial sharing (e.g., `KM`, `SC`). As a result, the host MMU PT-walk contention is reduced.

F. UVM-Driver Handled Far Faults

We implement and evaluate Trans-FW on UVM-driver based far fault handling. The fault handling is modeled the same as described in Section II-B. We implement Forwarding Table in CPU memory and use UVM-driver to handle the FT table lookup. Figure 26 shows the performance of Trans-FW on UVM-driver handled far faults in four GPUs. The results are normalized to the UVM-driver handled far faults. One can observe that, Trans-FW achieves an average of 68.6% performance improvement over the corresponding driver-based far fault handling. This is because our approach i) bypasses the unnecessary PT-walk in the GMMU caused by far faults, and ii) mitigates the UVM-driver handling contention, thereby reducing the far faults handling overhead. The results indicate that optimizations brought by Trans-FW also take effect in UVM-driver based multi-GPU systems.

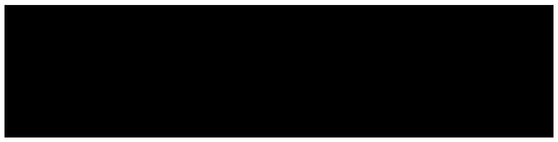
Fig. 26. Performance of Trans-FW on UVM-driver handled far faults normalized to UVM-driver based far faults handling.

G. Adopting Large-sized Pages

Fig. 27. Performance of Trans-FW with 2MB page normalized to baseline with 2MB page.

We next evaluate Trans-FW with 2MB page size. Figure 27 shows the performance of Trans-FW normalized to the baseline execution with 2MB page size. Trans-FW achieves an average of 38.6% performance improvement. This is less compared to the 4KB page size results. The reason is that, in the baseline execution, choosing a large page size effectively increases the L2 TLB hit rate, thus mitigating the contention in GMMU PT-walk. However, it is important to note that Trans-FW still achieves substantial performance improvements. This is because, by short-circuiting the PT-walk in GMMU and the PT-walk in the host MMU, Trans-FW is able to further reduce the PT-walk latency. Besides, although adopting large-sized pages reduces local page faults, it increases false sharing and causes extra inter-GPU page faults when a large page is frequently shared among different GPUs.

H. Comparison to PW-cache Prefetching

Fig. 28. Comparison to ASAP [46] PW-cache prefetching.

PW-cache prefetching is another well-known technique to improve the address translation efficiency by reducing the number of page walks. We compare Trans-FW with the state-of-the-art *ASAP* [46] address translation prefetching. The ASAP prefetching focuses on prefetching lower levels (i.e., L2) of page table during page table walk. In our comparison experiments, we employ ASAP in both GMMU PT-walk and host MMU PT-walk. Figure 28 shows the performance of Trans-FW and Trans-FW+ASAP normalized to the ASAP. We make the following observations. First, Trans-FW outperforms ASAP by an average of 38.4%. The reason is that ASAP does not mitigate PT-walk contention nor page fault handling latency. In contrast, our Trans-FW short circuits PT-walk caused by local page faults, mitigates the contention in PT-walk, and reduces the latency of handling page faults. Second, our approach can be combined with PW-cache prefetching to further improve the address translation efficiency. Specifically, Trans-FW+ASAP achieves 45.2% improvement over the ASAP. This is because the PW-cache miss latency is further reduced when combined Trans-FW and ASAP. In summary, the results demonstrate that the proposed Trans-FW is flexible to work with PW-cache prefetching strategies.

I. Combined with TLB Optimization

Fig. 29. Performance of Trans-FW when combined with Least-TLB normalized to Least-TLB [42].

To show the proposed Trans-FW can achieve better performance when combined with prior TLB optimizations, we apply Trans-FW to one of the state-of-the-art TLB optimizations – least-TLB [42]. Least-TLB focuses on improving TLB reach by eliminating translation redundancy and mitigating TLB contention in multi-GPU. Figure 29 shows that Trans-FW+least-TLB achieves a 57.9% performance improvement compared to using the least-TLB alone. In short, we demonstrate that Trans-FW is complementary to TLB optimizations and can bring additional performance benefits.

J. ML application

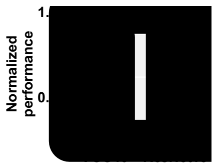

Fig. 30. Performance of Trans-FW with ML applications.

We also evaluate Trans-FW using real-world machine learning models (VGG16 and ResNet18) in data parallelism training. The results in Figure 30 indicate that Trans-FW achieves a performance improvement of 21.4% on VGG16 and 34.6% on ResNet18 normalized to their baseline executions. This demonstrates that the proposed Trans-FW also works in real multi-GPU training scenarios.

VI. Related work

TLB optimizations: The literature of computer architecture research has substantial prior works focusing on optimizing TLB performance in CPUs, GPUs, and accelerators [4], [5], [14], [18], [19], [37], [61]. These works can be divided into software-oriented TLB optimizations [10], [17], [23], [40], [41], [78] and hardware-oriented optimizations [12], [62], [63], [67], [74], [79]. Jaleel et al. [36] investigated translation memorization in DRAM and leveraged the last-level cache to keep the TLB entries. Tang et al. [71] improved the TLB reach in GPU through hardware-supported compression. Ausavarungnirun et al. [13] leveraged L2-TLB bypassing to reduce thrashing for concurrent multi-application executions. Compared to all these TLB optimization efforts, our approach focuses on optimizing page table walking in multi-GPUs, which is a more important performance bottleneck. Additionally, our approach is orthogonal or complementary to most TLB optimization and can bring further improvements. To this end, we propose Trans-FW. Figure 9 shows the high-level overview of our design. We evaluated our work with a state-of-the-art TLB optimization [42]. The results indicate that our approach can further improve the address translation efficiency in multi-GPU executions.

Page table walk optimizations: To accelerate the page table walk, prior works have explored the designs of MMU cache, page migration, and prefetching techniques [3], [14], [15], [25], [45], [46], [49], [55], [58], [66]. Pratheek et al. [64] proposed a dynamic page walk stealing to reduce PTW contention under multi-tenancy. Bhattacharjee [20] proposed a coalescing and sharing technique to reduce MMU cache misses. Achermann et al. [1] leveraged page table replication and migration to mitigate NUMA effects on page table walks. Most of these works focus on single CPU or single GPU execution. In contrast, our approach targets multi-GPU execution. We reveal that page sharing in multi-GPU execution is the root source causing PT-walk queuing and local page faults. The proposed approach leverages frequent page sharing and eliminates the performance burdens. Moreover, as we discussed in Section V-H, our approach is compatible with existing works (e.g., PW-cache prefetching) in single GPU to further improve the address translation efficiency.

VII. Conclusion

In this paper, we comprehensively and systematically studied the page table walk efficiency in multi-GPU executions. Our characterization indicates that there are three major latency bottlenecks in the page table walk. We proposed Trans-FW that leverages page sharing and remote forwarding to dynamically and automatically short circuit the page table walk in both GMMU and host MMU in multi-GPUs. Our experimental results show that Trans-FW significantly improves the overall performance by an average of 53.8%. We also compared with different PW-cache structure, page replication, large page, PW-cache prefetching, and TLB optimization to indicate the scalability, flexibility, and compatibility of the proposed design.

Acknowledgment

The authors would like to thank the anonymous HPCA reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2011146, #2154973, #1725657, #1910413, and a startup funding from the University of Pittsburgh.

References

[1] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, “Mitosis: Transparently self-replicating page-tables for large-memory machines,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 283–300.

[2] N. Agarwal, D. Nellans, M. Stephenson, M. O’Connor, and S. W. Keckler, “Page placement strategies for gpus within heterogeneous memory systems,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 607–618.

[3] N. Agarwal and T. F. Wenisch, “Thermostat: Application-transparent page management for two-tiered main memory,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 631–644.

[4] J. Ahn, S. Jin, and J. Huh, “Revisiting hardware-assisted page walks for virtualized systems,” in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 476–487.

[5] J. Ahn, S. Jin, and J. Huh, “Fast two-level address translation for virtualized systems,” *IEEE Transactions on Computers*, vol. 64, no. 12, pp. 3461–3474, 2015.

[6] T. Allen and R. Ge, “Demystifying gpu uvm cost with deep runtime and workload analysis,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 141–150.

[7] T. Allen and R. Ge, “In-depth analyses of unified virtual memory system for gpu accelerated computing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[8] AMD. (2015) AMD APP SDK OpenCL Optimization Guide.

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

[10] N. Amit, “Optimizing the TLB shootdown algorithm with page access tracking,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 27–39. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[11] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, “Mcm-gpu: Multi-chip-module gpus for continued performance scalability,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 320–332, 2017.

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

[13] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, “Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: ACM, 2018, pp. 503–518. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173169

[14] T. W. Barr, A. L. Cox, and S. Rixner, “Translation caching: Skip, don’t walk (the page table),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA ’10. New York, NY, USA: ACM, 2010, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970

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

[16] T. Baruah, Y. Sun, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, “Valkyrie: Leveraging inter-tlb locality to enhance gpu performance,” in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 455–466. [Online]. Available: https://doi.org/10.1145/3410463.3414639

[17] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, “Efficient virtual memory for big memory servers,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 237–248. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485943

[18] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee, “Scalable distributed last-level tlbs using low-latency interconnects,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 271–284.

[19] A. Bhattacharjee and M. Martonosi, “Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors,” in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 29–40.

[20] A. Bhattacharjee, “Large-reach memory management unit caches,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 383–394.

[21] L. Caucci and L. R. Furenlid, “GPU programming for biomedical imaging,” in *Medical Applications of Radiation Detectors V*, H. B. Barber, L. R. Furenlid, and H. N. Roehrig, Eds., vol. 9594, International Society for Optics and Photonics. SPIE, 2015, pp. 79 – 93. [Online]. Available: https://doi.org/10.1117/12.2195217

[22] I. Corporation. (2008) Tlbs, paging-structure caches and their invalidation.

[23] G. Cox and A. Bhattacharjee, “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*, ser. ASPLOS ’17. New York, NY, USA: ACM, 2017, pp. 435–448. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037704

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

[25] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, “Traffic management: a holistic approach to memory placement on numa systems,” *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 381–394, 2013.

[26] S. Dong and D. Kaeli, “Dnnmark: A deep neural network benchmark suite for gpus,” in *Proceedings of the General Purpose GPUs*, 2017, pp. 63–72.

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

[28] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo filter: Practically better than bloom,” in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 75–88. [Online]. Available: https://doi.org/10.1145/2674005.2674994

[29] D. Foley and J. Danskin, “Ultra-performance pascal gpu and nvlink interconnect,” *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[30] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, “Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 451–461.

[31] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, “Biomedical image analysis on a cooperative cluster of gpus and multicores,” in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: ACM, 2014, pp. 413–423. [Online]. Available: http://doi.acm.org/10.1145/2591635.2667189

[32] B. Hyun, Y. Kwon, Y. Choi, J. Kim, and M. Rhu, “Neummu: Architectural support for efficient address translations in neural processing units,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1109–1124. [Online]. Available: https://doi.org/10.1145/3373376.3378494

[33] Intel. (2017) 5-Level Paging and 5-Level EPT. [Online]. Available: https://mobt3ath.com/uplode/books/book-51381.pdf

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

[35] J. Andrew Rogers. (2015) MetroHash: Faster, Better Hash Functions. [Online]. Available: https://github.com/jandrewrogers/MetroHash

[36] A. Jaleel, E. Ebrahimi, and S. Duncan, “Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3309710

[37] G. B. Kandiraju and A. Sivasubramaniam, “Going the distance for tlb prefetching: an application-driven study,” in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 195–206.

[38] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, “Redundant memory mappings for fast access to large memories,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: ACM, 2015, pp. 66–78. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749471

[39] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, *Batch-Aware Unified Memory Management in GPUs for Irregular Workloads*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1357–1370. [Online]. Available: https://doi.org/10.1145/3373376.3378529

[40] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, “Latr: Lazy translation coherence,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: ACM, 2018, pp. 651–664. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173198

[41] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and efficient huge page management with ingens,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16. Berkeley, CA, USA: USENIX Association, 2016, pp. 705–721. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026931

[42] B. Li, J. Yin, Y. Zhang, and X. Tang, “Improving address translation in multi-gpus via sharing and spilling aware tlb design,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1154–1168.

[43] X. Long, X. Gong, and H. Zhou, “Deep learning based data prefetching in cpu-gpu unified virtual memory,” *arXiv preprint arXiv:2203.12672*, 2022.

[44] X. Long, X. Gong, and H. Zhou, “An intelligent framework for oversubscription management in cpu-gpu unified memory,” *arXiv preprint arXiv:2204.02974*, 2022.

[45] Z. Ma, Y. Tan, H. Jiang, Z. Yan, D. Liu, X. Chen, Q. Zhuge, E. H.-M. Sha, and C. Wang, “Unified-tp: A unified tlb and page table cache structure for efficient address translation,” in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 255–262.

[46] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, “Prefetched address translation,” in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: ACM, 2019, pp. 1023–1036. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358294

[47] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, “Beyond the socket: Numa-aware gpus,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 123–135.

[48] S. Mittal and J. S. Vetter, “A survey of cpu-gpu heterogeneous computing techniques,” *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[49] H. Muthukrishnan, D. Lustig, D. Nellans, and T. Wenisch, “Gps: A global publish-subscribe model for multi-gpu memory management,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 46–58.

[50] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, “Understanding pcie performance for end host networking,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 327–341.

[51] Nikolay Sakharnykh. (2017) Unified Memory on Pascal and Volta. [Online]. Available: http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf

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

[53] NVIDIA. (2022) NVIDIA Linux Open GPU Kernel Module Source. [Online]. Available: https://github.com/NVIDIA/open-gpu-kernel-modules

[54] NVIDIA Corp. (2018) Everything you need to know about unified memory. [Online]. Available: https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[55] A. Panwar, R. Achermann, A. Basu, A. Bhattacharjee, K. Gopinath, and J. Gandhi, “Fast local page-tables for virtualized numa servers with vmitosis,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 194–210.

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

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

[58] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, “Every walk’sa hit: making page walks single-access cache hits,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 128–141.

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

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

[61] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “Colt: Coalesced large-reach tlbs,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. USA: IEEE Computer Society, 2012, p. 258–269. [Online]. Available: https://doi.org/10.1109/MICRO.2012.32

[62] B. Pichai, L. Hsu, and A. Bhattacharjee, “Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[63] J. Power, M. D. Hill, and D. A. Wood, “Supporting x86-64 address translation for 100s of gpu lanes,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 568–578.

[64] B. Pratheek, N. Jawalkar, and A. Basu, “Improving gpu multi-tenancy with page walk stealing,” in *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[65] J. Ryoo, M. Fan, X. Tang, H. Jiang, M. Arunachalam, S. Naveen, and M. T. Kandemir, “Architecture-centric bottleneck analysis for deep neural network applications,” in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 205–214.

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

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

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

[69] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, “Mgpusim: Enabling multi-gpu performance modeling and optimization,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 197–209. [Online]. Available: https://doi.org/10.1145/3307650.3322230

[70] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, R. Ubal, X. Gong, S. Treadway, Y. Bao, V. Zhao, J. L. Abellán *et al.*, “Mgsim+ mg-

mark: A framework for multi-gpu system research," *arXiv preprint arXiv:1811.02884*, 2018.

[71] X. Tang, Z. Zhang, W. Xu, M. T. Kandemir, R. Melhem, and J. Yang, "Enhancing address translations in throughput processors via compression," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 191–204. [Online]. Available: https://doi.org/10.1145/3410463.3414633

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

[73] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, "Exploiting page table locality for agile tlb prefetching," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 85–98.

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

[75] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, 2018, pp. 41–53.

[76] J. Wei, J. Lu, Q. Yu, C. Li, and Y. Zhao, "Dynamic gmmu bypass for address translation in multi-gpu systems," in *Network and Parallel Computing*, X. He, E. Shao, and G. Tan, Eds. Cham: Springer International Publishing, 2021, pp. 147–158.

[77] C. Xie, F. Xin, M. Chen, and S. L. Song, "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*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 53–65. [Online]. Available: https://doi.org/10.1145/3307650.3322247

[78] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 698–710. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322223

[79] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering translation bandwidth with virtual caching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 113–127. [Online]. Available: https://doi.org/10.1145/3173162.3173195

[80] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining hw/sw mechanisms to improve numa performance of multi-gpu systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 339–351.