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

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

Yanan Guo
University of Pittsburgh
Pittsburgh, PA, USA
yag45@pitt.edu

Yueqi Wang
University of Pittsburgh
Pittsburgh, PA, USA
yuw249@pitt.edu

Aamer Jaleel
NVIDIA
Westford, MA, USA
ajaleel@nvidia.com

Jun Yang
University of Pittsburgh
Pittsburgh, PA, USA
juy9@pitt.edu

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

ABSTRACT

Multi-GPU systems have emerged as a desirable platform to deliver high computing capabilities and large memory capacity to accommodate large dataset sizes. However, naively employing multi-GPU incurs non-scalable performance. One major reason is that execution efficiency suffers expensive address translations in multi-GPU systems. The data-sharing nature of GPU applications requires page migration between GPUs to mitigate non-uniform memory access overheads. Unfortunately, frequent page migration incurs substantial page table invalidation overheads to ensure translation coherence. A comprehensive investigation of multi-GPU address translation efficiency identifies two significant bottlenecks caused by page table invalidation requests: (i) increased latency for demand TLB miss requests and (ii) increased waiting latency for performing page migrations. Based on observations, we propose IDYLL, which reduces the number of page table invalidations by maintaining an "in-PTE" directory and reduces invalidation latency by batching multiple invalidation requests to exploit spatial locality. We show that IDYLL improves overall performance by 69.9% on average.

CCS CONCEPTS

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

KEYWORDS

multi-GPU, page table invalidation, page sharing

ACM Reference Format:
Bingyao Li, Yanan Guo, Yueqi Wang, Aamer Jaleel, Jun Yang, and Xulong Tang. 2023. IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3614269

1 INTRODUCTION

Multi-GPU systems (such as NVIDIA's DGX [48] and DGX-2 [49], Intel Xe [30]) have become the preferred platform in achieving scalable performance for real-world applications [20, 25, 26, 28, 33, 42, 71, 79, 82]. Multi-GPU systems generally employ unified virtual memory (UVM) to simplify the programming across GPUs [51, 63]. Despite its advantages, UVM performance highly depends on efficient address translation which involves hierarchical TLB lookups and GPU local page table walks. During the translation process, one major performance penalty is page *far fault* which is generated when the page is not present in GPU local memory. The number of far faults can be significant, especially when multiple GPUs frequently share pages. To reduce the number of far faults and enable efficient data sharing among GPUs, modern multi-GPU systems (such as Nvidia Ampere GPUs [18]) employ remote mapping as well as counter-based page migrations. Specifically, a GPU can maintain the address translation of a remote page (i.e., a page residing in another GPU's local memory) in its local page table. When the GPU requests this remote page, the request is routed to the remote GPU and the requested data is supplied by the remote GPU. Depending on the intensity of remote access, a page can be migrated across GPUs to enhance the page access locality. This is realized by implementing a page counter and the migration is determined by thresholds [27, 51, 63]. When a page migration occurs, existing page table entries (PTEs) on other GPUs mapping to the same page must be *invalidated* to ensure translation coherence. The invalidations are broadcasted to all GPUs for consistency. When pages are frequently accessed from different GPUs, frequent page migration incurs a significant number of invalidations in each GPU. The invalidations contend with existing demand TLB miss requests, therefore severely degrading application performance.

While many prior studies have explored GPU address translation optimizations and page migration optimizations, we are unaware of any studies that investigate the effects of frequent page migration invalidations in multi-GPU systems. We provide a comprehensive summary of the prior art and compare our approach with them in Table 1. First, prior TLB optimizations include translation clustering [56], page coalescing [57], address compression [70], and TLB prefetching [9, 74]. While these techniques effectively improve TLB reach and reduce far faults, they help little with the page sharing caused page migration. Specifically, whenever a page migrates, the translation changes and a TLB shootdown is necessary

Techniques	Reduce invalidation	Improve PTW	Write intensive	Multi -GPU
TLB optimizations [31, 77] [11, 54, 56, 57, 70, 74]	No	Yes	No	No
PW-cache design [14, 43, 44, 55]	Partial	Yes	Yes	No
Page walk scheduling [61, 65]	No	Yes	Yes	No
Large page [7, 53, 58]	Partial	Partial	No	No
Dynamic page migration [10]	No	No	Yes	Yes
Page replication [22, 47]	Yes	No	No	No
Our approach	Yes	Yes	Yes	Yes

Table 1: Comparison with prior techniques.

to ensure correctness [5]. Second, existing page walk cache optimizations [14, 43, 44, 55] can accelerate the page table walks for invalidation requests by reducing page walk cache misses. However, the substantial invalidations thrash the page walk cache and lead to frequent eviction of useful entries required by existing demand TLB miss requests. Third, page walk scheduling works [61, 65] enable a trade-off between page table walk throughput and fairness. However, these works are not applicable to page migration invalidations as they do not help with the significant amount of invalidations generated by intensive page sharing across GPUs. Fourth, a large page [7, 53, 58] increases the TLB reach and reduces the contention in page table walk in a single GPU. However, in multi-GPU, when a large page is frequently shared among different GPUs, the false sharing may introduce more page migrations and additional invalidation requests. Fifth, the dynamic page migration [10] is effective in reducing remote data access, however, the PTE invalidation overheads caused by page migration have yet to be addressed. Finally, page replication [22, 47] enables pages to be accessed locally without page migration, therefore reducing the number of invalidations. However, it is not feasible for applications where the shared pages have read-write properties, as write operations still require invalidating PTEs and pages. Further, with substantial page sharing among multiple GPUs, page replication is not scalable [40, 62]. In a nutshell, none of the prior works investigate the effect of invalidations in UVM-managed multi-GPU.

This paper quantitatively shows that contention between demand TLB miss requests and page migration-induced page table invalidation requests significantly limits multi-GPU performance. We propose **I**n-PTE **D**irector**Y** and **L**azy Inva**L**idation (IDYLL) which employs two key mechanisms to minimize this contention. First, we reduce the number of unnecessary invalidation requests by employing an "in-PTE" directory that leverages unused bits in the page table entry to record which GPUs hold valid address mappings for the corresponding PTE. Second, we minimize interference between invalidation requests by employing lazy invalidation which exploits spatial locality in page table updates by batching multiple invalidation requests with nearby virtual addresses. We design a hardware structure called Invalidation Request Merging Buffer (IRMB) to temporarily hold these batched invalidation requests and lazily write them back to the page table. Specifically, the IRMB is checked in parallel with the GPU L2 TLB lookup. For those TLB misses, if they are found in the IRMB, the corresponding page table walks are removed to avoid accessing stale PTEs and to ensure the correctness of the translation.

The major contributions of this work include:

- We show that a major performance bottleneck in UVM-managed multi-GPU is due to a significant amount of page table invalidations. We provide a detailed analysis of how the page table invalidations affect demand TLB miss requests and page migration waiting latency.
- We propose IDYLL, a software-hardware co-design to mitigate the page table invalidation overhead and improve overall application performance. IDYLL maintains a software-managed "in-PTE" directory that filters unnecessary invalidation requests and a lightweight hardware-managed structure to temporarily buffer invalidation requests, removing them from the contention of page table walk resources while maintaining execution correctness.
- We show that IDYLL improves overall performance by 69.9% on average across our suite of multi-GPU applications. We show that IDYLL outperforms large pages, page replication, and page table walk optimization. IDYLL is orthogonal to these approaches and can be combined with them to further improve performance.

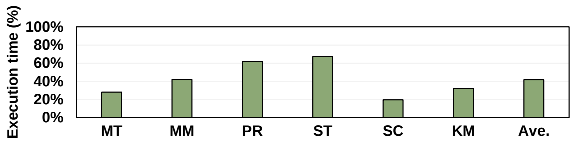

Figure 1: Page table invalidation overhead.

2 MOTIVATION

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

Page migration modifies the existing virtual to physical mapping. Consequently, all system-wide data structures (i.e., per-GPU TLBs and per-GPU local page tables) containing the old virtual-to-physical mapping must be invalidated. To perform this task, conventional UVM drivers simply broadcast page table invalidation requests to all GPUs in the system (even those GPUs that never requested the translation). In response, GPUs receiving the page table invalidation requests walk their respective local page table and invalidate the corresponding PTE (even if it were invalid to begin with). A detailed background on step-by-step multi-GPU address translation procedure with invalidations is given in Section 3.2.

Broadcasting page table invalidation requests works well when page migrations are infrequent. However, when applications exhibit high page sharing between multiple GPUs (quantitatively analyzed in Section 5.1), frequent page migrations significantly increase the number of page table invalidation requests to invalidate GPU local PTEs. These invalidations contend with existing demand TLB misses thereby increasing their miss latency. Demand TLB miss latency is performance critical and must be reduced to improve GPU performance.

Figure 1 demonstrates the page table invalidation overheads for representative multi-GPU applications[1] running on a 2-GPU NVIDIA A100 system. We use *uvm-eval* [73] to profile the GPU page table invalidation time. The figure shows that nearly half (average 42%) of total execution time is spent on handling page table invalidations. Applications with high page sharing (e.g., PR

[1]These are multi-GPU ready applications with large input sets running on real hardware and are compatible with *uvm-eval* [73].

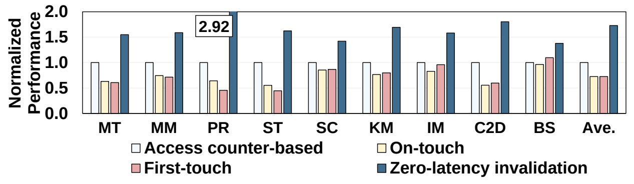

Figure 2: Performance of each scheme relative to access counter-based migration.

and ST) tend to have higher page invalidation overheads. These results are consistent with previous GPU UVM studies [3, 16] and strongly motivate the need for substantially reducing the page invalidation overheads.

To illustrate the performance opportunity from eliminating page invalidation overheads altogether, Figure 2 presents a simulation-based study that depicts the performance of an ideal page migration policy that incurs no page table invalidation overheads. That is, the per-GPU page table invalidation requests incur *zero* latency and bandwidth contention with the PTEs being updated instantaneously. We use an industry-validated multi-GPU simulation framework (MGPUsim [68], details in Section 4). Our simulated baseline system employs *access counter-based page migration* (which is the baseline on NVIDIA A100 GPUs [18]), where pages are migrated only when remote accesses reach a given threshold. The figure shows that an idealized system with zero page table invalidation overheads outperforms the baseline system by 38%-1.92× (average 73%). For reference, we also report the performance of (a) *first-touch migration* where the page is pinned to the GPU that first accessed a page with no further migrations (b) *on-touch migration* where pages are always migrated to the requesting GPU. While both these page migration policies can have lower page table invalidation overheads, we observe that they generally perform worse than the access counter-based migration policy either because of the increased number of remote memory accesses or frequent page migrations.

The results from Figure 1 and Figure 2 clearly demonstrate the importance of reducing page table invalidation overheads. Such overheads stem from page table walk contention between demand TLB miss requests and page table invalidation requests. Reducing page invalidation overheads can be accomplished by (i) *reducing the number of invalidations sent* by employing a directory to only send page table invalidation requests to GPUs that hold the corresponding address translation (ii) *reducing the invalidation latency* by batching invalidation requests to exploit spatial locality between multiple invalidation requests. Before we discuss our solution, we provide a detailed discussion of address translations in multi-GPU systems (Section 3) and quantitatively investigate the contentions caused by page table invalidations (Section 5).

3 BACKGROUND

3.1 Multi-GPU Architecture

In this work, we focus on UVM-managed discrete multi-GPU systems where the GPUs have unified virtual memory address space and use pointers to access memory in remote GPUs. The GPUs are connected using a high-bandwidth interconnect such as PCIe or NVLink. Figure 3 shows the architecture details of the targeted baseline multi-GPU system. Each GPU consists of multiple Shader

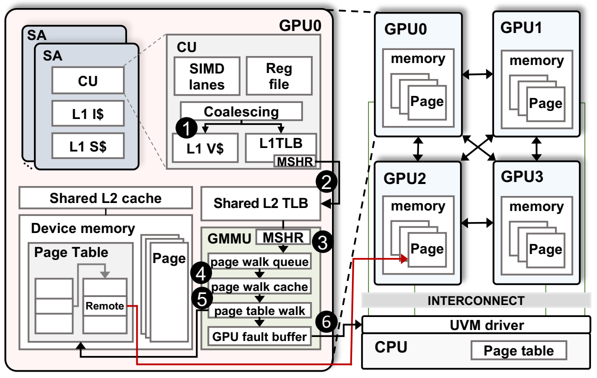

Figure 3: Baseline GPU architecture.

Arrays (SAs), each of which further contains multiple compute Units (CUs), a.k.a., SMs in NVIDIA terminology. Every CU has its own private L1 data cache (L1V$); all CUs within an SA share the L1 scalar cache (L1S$) and the L1 instruction cache (L1I$). There is also a larger unified L2 cache that is shared among all the SAs. Each GPU also features a multi-level TLB hierarchy to accelerate address translation: each CU has a private fully associative L1 TLB, and all CUs (in all SAs) share the L2 TLB. A discrete GPU usually has its own local memory as well as a local page table. The GPU Memory Management Unit (GMMU) handles GPU page table walks. A GMMU typically consists of (i) a page walk queue for buffering the translation requests, (ii) a page walk cache holding the entries of page table levels used in recent translation requests, and (iii) multi-threaded page table walker that handles multiple translation requests concurrently. In UVM-managed multi-GPU systems, the UVM driver on the CPU side is responsible for handling all GPU far faults: the UVM driver maintains a centralized page table holding the up-to-date address translations for all GPUs.

3.2 Address Translation in Multi-GPU

The address translation process is also illustrated in Figure 3. Upon a memory request, the L1 cache and the L1 TLB lookups are performed in parallel, assuming the L1 cache is virtually indexed and physically tagged (❶). If the request misses in the L1 TLB, it first checks the L1 Miss Status Holding Register (MSHR), and the request is sent to the L2 TLB for lookup upon MSHR miss (❷). Similarly, requests missing in the L2 TLB are further sent to the GMMU (❸) to perform page table walks. The request is temporarily stored in the page walk queue if there are no available page walk threads. When performing a page walk, GMMU generates several memory requests to access the local page table (❺). To take advantage of the temporal locality of the page table, the GMMU maintains page walk cache for each level of the page table (❹). If the page walk fails, a far fault is propagated to the GMMU and held in the GPU Fault Buffer [2, 3]. Then, the GMMU notifies the host UVM driver about the far fault by generating an interrupt (❻). The UVM driver, on the host side, fetches the fault information, groups faults into batches, and caches it on the host (the batch size is 256 [50]), and later resolves the fault using the centralized page table. After these steps, it initiates the target data transfer and updates the requesting GPU's local page table. Eventually, the GPU re-performs the address translation after the far fault is resolved.

Remote Mapping: In a multi-GPU system, a GPU can access the memory of a remote GPU [16]. This is realized by allowing the local page table to store the address that maps virtual pages to physical pages residing in remote GPU's memory. Specifically, when a GPU is accessing a page in the memory of another GPU for the first time, it generates a far fault since its local page table does not yet contain the translation information. Then, this far fault is passed to the UVM driver and the driver sends the translation result (along with some other information of the page) to the requesting GPU. After this, the GPU updates the local page table, as well as fetches the data at cacheline granularity from the remote GPU. Upon receiving the data from the remote GPU, the data is directly sent to the CU but is *not* cached in the cache hierarchy [24]. Consequently, future GPU accesses to the remote data can be translated by the local page table and the data will be serviced by the remote GPU.

3.3 Page Migration Scheme

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

On-touch migration: Every time a GPU accesses a page residing in another GPU, the page is migrated into the requesting GPU memory. While it guarantees page accesses are going to local pages, frequent "ping-pong" migration may decrease the performance. This is the "On-touch" bar in Figure 2.

First-touch migration: A page is migrated from the CPU to the GPU on the GPU's first access. After this initial migration, this page is pinned on that GPU, i.e., it will never be migrated to another GPU. Compared to on-touch migration, first-touch migration does not incur frequent page migrations and thus avoids the invalidation/migration overheads. However, remote memory accesses may incur high latency. This is the "First-touch" bar in Figure 2.

Access counter-based migration: Recent Nvidia GPUs (Volta and newer generations [63]) use page access counters to delay the migration of pages. On each remote memory access, the corresponding page access counter is incremented. A page migration is only performed when this counter reaches a certain threshold (e.g., 256 [50]). There are four steps for access counter-based page migration. First, the GPU initiates a migration request to the UVM driver. Second, the UVM driver broadcasts the invalidation requests to every GPU in the system since the UVM driver is unaware of which GPU(s) have the translation of this page. Third, upon receiving invalidation requests, each GPU performs TLBs shootdown and invalidates the page table entries to ensure translation coherence [10]. The PTE invalidation is performed in a way similar to the conventional address translation procedure: the GMMU starts a page walk which may contend with other page walk requests. Finally, the UVM driver initiates the data transfers. This scheme incurs fewer remote memory accesses compared to first-touch migration, and fewer page migrations compared to on-touch migration. This is the bar labeled "Access counter-based" in Figure 2.

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
Page table walk	GMMU 8 shared page table walker [61, 65, 75], 100-cycle latency per level [29]
Page walk cache	128 entries shared across page table walker [61]
Page walk queue	64 entries
Access counter threshold	256 [50]
Inter-GPU network	300GB/s NVLink-v2
CPU-GPU network	32GB/s PCIe-v4

Table 2: Baseline multi-GPU configuration.

Abbr.	Application	Benchmark Suite	MPKI	Access Pattern
KM	KMeans	Hetero-Mark	50.67	Adjacent
PR	PageRank	Hetero-Mark	78.21	Random
BS	Bitonic Sort	AMDAPPSDK	3.42	Random
MM	Matrix Multiplication	AMDAPPSDK	11.21	Scatter-Gather
MT	Matrix Transpose	AMDAPPSDK	185.52	Scatter-Gather
SC	Simple Convolution	AMDAPPSDK	15.76	Adjacent
ST	Stencil 2D	SHOC	36.24	Adjacent
C2D	Convolution 2D	DNN-Mark	21.42	Adjacent
IM	Image to Column	DNN-Mark	18.31	Scatter-Gather

Table 3: List of applications.

4 METHODOLOGY

We use MGPUSim [68], a multi-GPU simulator that is validated against industrial GPU architecture.

Baseline GPU configuration: We target a 4-GPU system where each GPU has its local page table. The baseline configurations are listed in Table 2. In the baseline, we assume 4KB page size and leave the study of large page to Section 7.2. In all experiments, the CTA scheduling policy is (i) round-robin for CUs within a GPU and (ii) greedy across GPUs which considers intra-GPU locality. This scheduling approach captures the GPU inter-CTA locality and also ensures the computing balance across CUs. We use access counter-based migration policy in both baseline and our approach.

Applications: We use several representative applications from various benchmark suites (see Table 3) including Hetero-Mark [69], AMDAPPSDK [4], SHOC [21], and DNN Mark [23]. We use the multi-GPU implementations of these applications available from [68], which is also widely adopted and evaluated in prior works [10, 41, 62]. Note that, the applications cover different data access/sharing patterns in multi-GPU execution. Specifically, (PR, BS) exhibit random access pattern where each GPU can generate reads and writes to any other GPU in an unpredictable manner. On the other hand, (KM, SC, ST, C2D) exhibit adjacent access pattern where the input data is batched and shared with the neighboring GPUs. Finally, (MM, MT, IM) exhibit scatter-gather access pattern where each GPU stores a fraction of input and output matrices, and each GPU reads/writes data from local/remote GPUs. Since different sharing behaviors bring different intensities of address invalidations, these representative sharing patterns allow us to evaluate how our design is generic to a wide spectrum of applications.

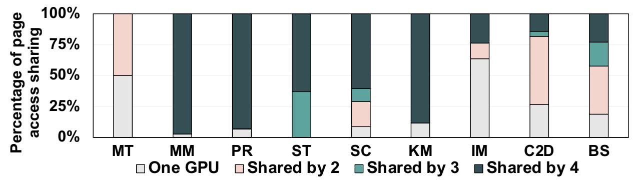

Figure 4: Distribution of accesses referencing shared pages.

We also provide the L2 TLB misses per kilo instructions (MPKI) in Table 3.

5 PAGE SHARING AND OVERHEADS

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

5.1 Multi-GPU Page Sharing Characterization

In a multi-GPU system, PTE invalidations are triggered during page migration when a page is frequently referenced by multiple GPUs. As such, we first investigate the page sharing behavior of applications in a multi-GPU environment. Page sharing in multi-GPU applications is common as threads running on different GPUs may access the same data structures. In this paper, we define the page access sharing ratio as the ratio of total shared page accesses to the total accesses. Figure 4 shows that there exists significant page sharing among multiple GPUs. For example, in MM, PR and KM, almost all accesses are to pages shared by all GPUs. In MT, C2D, BS, a large fraction of accesses is concentrated on the pages that are shared by 2 GPUs. This is because, PR has random access pattern where any GPU needs to both read and write data from/to the entire GPU address space. C2D needs to access input data from surrounding indices that are resident on other GPUs.

5.2 Page Table Walk Characterization

As described in Section 3.3, when a page needs to be migrated in the remote mapping approach, all GPUs need to perform local page table walks to invalidate their corresponding page table entries before the page can be migrated. Figure 5 provides the distribution of requests to the page walker in terms of PTE invalidation requests (necessary and unnecessary) and demand TLB miss requests. One can observe the number of invalidation requests accounts for a quarter (i.e., 27.2%) of total requests to the page walker. For applications with excessive page access sharing, they have a higher percentage of invalidation requests, such as MM, PR, and KM. This is because a large number of pages are heavily accessed by multiple GPUs, which causes a large number of page migrations and thus significant PTE invalidation requests.

We also show the percentage of unnecessary PTE invalidation requests in Figure 5. Specifically, since each migration broadcasts invalidation requests to all GPUs, those GPUs that have not accessed the page (or their corresponding PTEs are already invalid) still need to perform page table walks. This is because they do not have information on whether the local PTE is valid without actually walking the page table. We refer to these invalidations as unnecessary invalidation requests. On average, the results in the

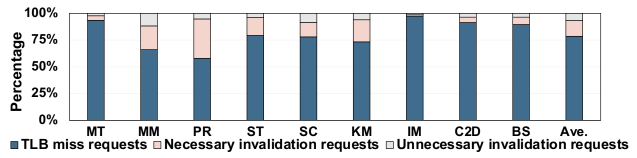

Figure 5: Percentage of invalidation requests and demand TLB miss requests.

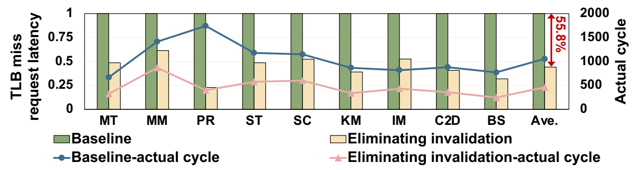

Figure 6: Demand TLB miss requests latency in baseline and zero-latency invalidation execution (normalized values and the averaged exact number of cycles).

figure indicate that nearly one-third (i.e., 32%) of PTE invalidations broadcasted are unnecessary.

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

Figure 6 assumes a hypothetical system where the PTE invalidation requests incur no contention to demand TLB requests. In such a system, when the GPU receives invalidation requests from the UVM driver, all latency and contention from invalidation requests are removed, including waiting in the page walk queue, page walk cache lookup, and performing page table walks. The figure shows the demand TLB miss request latency of such a system normalized to the demand TLB miss request latency in the baseline system with PTE invalidations. We also plot the actual values of averaged cycles shown as lines in the figure corresponding to the right y-axis. The result shows that without the interference of invalidation requests, the demand TLB miss request latency is on average reduced by 55.8% of the baseline. This is because invalidation requests follow the same page table walk process as demand TLB miss requests, which includes waiting in the page walk queue, lookup the page walk cache, and walking the page table after the page walk cache miss. Therefore, the invalidation requests (i) extend the page table walk queuing time for demand TLB miss requests, (ii) thrash the page walk cache, which may reduce the page walk cache hit rate for demand TLB miss requests, and (iii) increase the contention of page table walk threads in the GMMU.

Once a page is determined to migrate, those requests accessing that page will need to wait for the page migration to complete and establish new translation mapping before those requests can access the page. Since the invalidation delays the page migration process (specifically, translation resolve time), it also extends the waiting latency of those requests. Figure 7 plots these extra page migration waiting latency caused by invalidation requests. We observe the extra waiting latency is 38.3% of the page migration

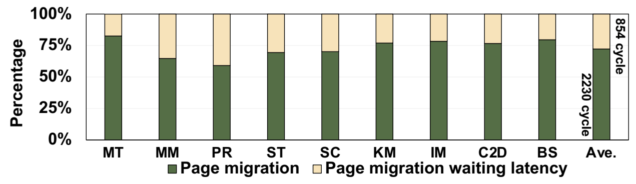

Figure 7: Percentage of page migration latency and page migration waiting latency.

latency. However, we want to mention that, not all these waiting latencies contribute to performance degradation because they are not on the critical path and can be hidden by the computation context switching in GPU. But also note that, for memory-intensive applications, there is not much computation to hide these latencies and they play a sizable role in the overall performance.

6 IN-PTE DIRECTORY AND LAZY INVALIDATION (IDYLL)

6.1 High Level Overview

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

6.2 In-PTE Directory Invalidation

Section 5.2 showed that a large fraction of PTE invalidations are unnecessary. We propose to filter the needless PTE invalidation requests using an In-PTE Directory Invalidation design on the host side. This approach avoids contention and queuing delays in the page walk queues, page walk caches and further reduces several operations on those GPUs that do not hold a valid translation to begin with. We now discuss our In-PTE Directory Invalidation design in detail.

Which GPUs to invalidate? The In-PTE Directory Invalidation should be aware of all translation mappings in each GPU and which GPU has these valid translation mappings. However, keeping track of such a high volume of entries can cause a lot of memory overhead. Fortunately, the host-side page table holds all valid and up-to-date address translations for all GPUs. We only need to add information of which GPUs hold these valid mappings to the corresponding address translation entries. To this end, we leverage unused bits

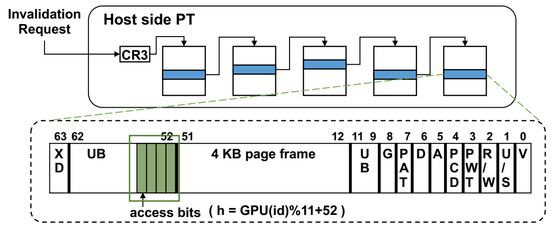

Figure 8: Page table entry format for 4 KB pages.

in the host-side PTEs as *access bits* for each GPU to store the GPU access information. Figure 8 shows the page table entry format in 4 KB pages[2]. Specifically, bits 51 to 12 store the physical page number for the virtual address, bits 62-52 and 11-9 are unused bits, and the remaining bits store metadata about the page. Note that, the maximum number of unused bits in the current PTE format is 14 bits. When the number of GPUs exceeds the maximum unused bits, the GPUs access bits cannot be mapped one-to-one with the unused bits. Therefore, we use a modular hash function to map multiple GPUs access bits to one unused bit. To simplify the calculation, we only use unused bits 62-52 for access bits. Specifically, the hash function is h(GPU_{id}) = GPU_{id} % m + 52, where m represents the number of unused bits used for access bits, which is equal to 11 in our design[3]. For example, in our default 4-GPU system, the unused bits 55-52 of PTE correspond to the access bit of GPU_3-GPU_0 (as shown in the green box of Figure 8). Mapping multiple GPU access bits into a single slot can lead to false positives, which only lead to unnecessary requests being sent to the GPU, but do not affect correctness. Nevertheless, the number of unnecessary requests sent to the GPU is significantly reduced compared to the baseline. Initially, all access bits in the host-side PTE are set to 0. When a GPU, for example, GPU_0, accesses a page for the first time, it will miss all local GPU TLBs and the local page table entry of this page is also invalid. A far fault is generated and sent to the host side. The host-side page table is then walked to get the desired address translation. At this point, the access bit for GPU_0 in the host-side PTE (the unused bit 52) is set to 1, as GPU_0 will establish a valid mapping to its page table when the address translation is replayed.

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

[2]Unused bits stay constant with different page sizes [15].
[3]We evaluate scalability with 4 unused bits in Section 7.2.

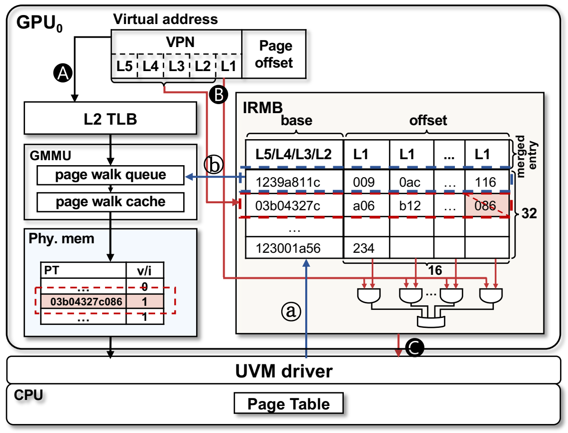

Figure 9: Overview of IDYLL.

and this additional latency is marginal to the overall performance. Nevertheless, we include these overheads in our later evaluation.

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

6.3 Lazy Invalidation

In-PTE Directory Invalidation eliminates unnecessary invalidations while helping little with those GPUs that hold the valid translations and must perform the page table invalidations. Due to the significant page sharing in multi-GPU, there are still substantial valid translation mappings that need to be invalidated when performing page migrations. These invalidation requests introduce extra latency for both existing demand TLB miss requests and page migration waiting time as specified in Section 5.2. Therefore, we design a lightweight hardware component called *Invalidation Request Merging Buffer* (IRMB) in each GPU. The IRMB acts as a "valve" for the invalidation requests by temporally buffering incoming invalidation requests and lazily updating the local page table with minimum impact to existing demand TLB miss requests. Note that, we only perform lazy updates to the PTE while keeping the TLB shootdown process as it is in the baseline. That is, upon receiving an invalidation request, the TLB is immediately invalidated.

Invalidation Request Merging Buffer (IRMB): The IRMB tracks the virtual page number (VPN) of invalidation requests that are received from the UVM driver. In particular, we observe that in many applications, pages being migrated are nearby to each other in the address space. Therefore, the virtual addresses (VAs) of invalidation requests are also nearby and share a significant number of identical bits in the addresses. Based on this observation, we design compressed entries for IRMB. The key idea behind this is to exploit the similarity of VAs where the VPN of invalidation requests in the IRMB at a certain period has a large number of identical bits. These identical bits can be merged so that IRMB can accommodate more invalidation requests. Besides, the nearby invalidations that are merged into one entry can leverage the same page walk cache when updating the invalidations to the page table to amortize the invalidation overheads. Therefore, in our design, we merge the invalidation request with the same high-level VPN into one entry. Figure 9 illustrates the microarchitecture of the IRMB. Specifically, the VPN of a page is partitioned into a 36-bit base (L5-L2 level of VA), and a 9-bit offset (L1 level of VA). The VPN with the same base coalesces into one *merged entry*. The IRMB consists of 32 merged entries with different bases, and each merged entry comprises 16 different offsets[4].

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

IRMB writeback: In our design, we use the IRMB to temporarily record the invalidation request first. It is crucial to propagate invalidations without introducing significant latencies to the critical path execution. Ideally, the invalidations should overlap with normal executions so that the invalidation overheads could be hidden. Therefore, first, in our design, when the page table walker is available, we invalidate the LRU merged entry corresponding PTEs and also evict this merged entry in the IRMB. In such a case, the invalidation will neither affect demand TLB miss requests nor page migration. Second, in the case of the IRMB being full, the propagation of invalidations is essential. Therefore, when an eviction happens in the IRMB, all PTEs corresponding to VAs in the evicted merged entry are invalidated sequentially. This allows the invalidations to be handled in a batch, and improves the L2 level page walk cache hit rate since all these invalidations have the same higher level of the VA.

IRMB lookup: Figure 9 also illustrates the lookup procedure in the IRMB. Specifically, When a translation request misses GPU L1 TLB, the L2 TLB (Ⓐ) and the IRMB (Ⓑ) are searched in *parallel*. Three different scenarios may happen. First, if the request hits the L2 TLB, the lookup in IRMB is abandoned, and the request is handled the same as the baseline. Second, if the request misses the L2 TLB and also misses the IRMB, which implies the desired PTE is up-to-date, whether valid or invalid. Therefore, the request behaves the same as the baseline, waiting in the page walk queue, looking up the page walk cache, and walking the page table. Third, if the request misses the L2 TLB but hits the IRMB, which indicates

[4]We also evaluate different IRMB sizes in Section 7.2.

that the translation mapping in the page table cannot be used as it should be invalidated. Then the request bypasses the local page table walk and directly raises a far fault to alert the UVM driver (ⓒ). By doing so, the latency of demand TLB miss requests that eventually cause far faults is further reduced. Note that the corresponding PTE invalidation process may also be bypassed in this case. This is because a far fault is generated, and a new translation mapping will be received after the host-side page table walk. Therefore, we can update the PTE directly after a new mapping is replayed without invalidating it. It can happen that the corresponding entry is evicted from the IRMB before the new mapping is received due to capacity conflicts. In this case, the evicted entries follow the IRMB eviction process and perform the PTE invalidation in the page table, even though these invalidations are unnecessary. Note that, before a new mapping is received, there won’t be any subsequent requests to the same page being sent to GMMU for page table walk. This is because the original request that triggers the new mapping resides in the L2 TLB MSHR. All subsequent requests to the same page will be blocked and held at the L2 TLB MSHR. Therefore, if the corresponding entry is not found in the IRMB and a new mapping is not received, it is guaranteed that the request does not access a stale translation. Note also that, upon receiving a new mapping, the IRMB is checked if the corresponding VPN is present in the IRMB. If it is not in the IRMB, the new mapping is directly inserted into the page table walk queue for PTE update. If it is found in the IRMB, the particular offset from the merged entry in IRMB is removed, since this translation mapping is established in the page table as a valid translation.

Correctness: The TLB is flushed immediately whenever this is a page migration in both baseline and our approach, we expect the security to be the same in both baseline and our approach. Our Lazy Invalidation will keep the stale entry in the page table and use IRMB to indicate it for execution correctness.

Overheads: In our configuration, the IRMB has a total of 32 merged entries. Each merged entry comprises 16 offsets and a base. The base is $4 \times 9 = 36$ bits, and the offset is $16 \times 9 = 144$ bits. Therefore, each merged entry occupies 180 bits. The total size of IRMB is $(36 + 144) \times 32/8 = 720$ bytes. We use CACTI [72] to estimate the areas and the results show that IRMB is 0.9% compared to the areas of GPU L2 TLB.

6.4 IDYLL-InMem: An Alternate Design

In the event that the unused bits in page table entry are reserved for other purposes (e.g., implementing custom memory management policies, setting access permissions) [37, 54, 67], we propose an alternative in-memory directory design, IDYLL-InMem, that tracks GPU translation residency for all pages in the system. The in-memory directory, referred to as the VM-Table, achieves the same functionality as the In-PTE Directory. Specifically, each entry in the VM-Table is 64 bits and stores VPN (45 bits) and GPU access bits (19 bits, all initialized to 0s). If the system has more than 19 GPUs, we employ the same hash function as the In-PTE Directory approach to hash the GPU access bits.

Having every translation to access the in-memory VM-Table for page residency involves memory accesses that incur additional memory bandwidth and latency overheads. To mitigate the overheads, we propose a hardware-managed cache (VM-Cache) to cache frequently-accessed entries from the VM-Table. The VM-Cache has 64 entries (4-way associative) and uses write allocate and write back policy. Figure 10 illustrates our proposed design.

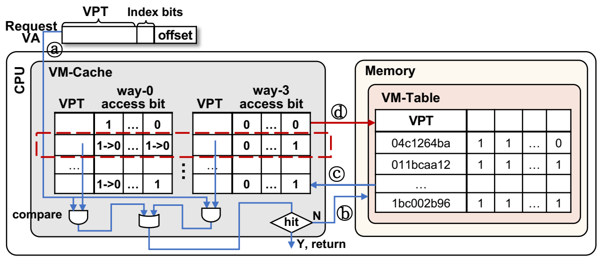

Figure 10: Overview of IDYLL-InMem.

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

Overheads: Each entry of the VM-Table is 8 bytes. Suppose the memory footprint of an application is 2^x, it needs $2^{(x-12)}$ entries in VM-Table. Therefore, the total space needed for the VM-Table would be $2^{(x-12)} \times 8\ bytes = 2^{(x-9)}$, which is only 0.2% of the application’s memory footprint. The space overhead of the VM-Table is negligible compared to the overall memory of the system. As for the overhead of VM-Cache, we use 64 entries. The hardware overhead would be $(41 + 19)\ bits \times 64\ entries = 480\ bytes$. We use CACTI to estimate the area of VM-Cache and the result shows it is 0.04% of L1 cache (32KB 8-associative CPU L1 cache) area.

7 EVALUATION

7.1 Overall Performance

Figure 11 shows the performance improvements of using Lazy Invalidation only, In-PTE Directory Invalidation only, and our proposed IDYLL normalized to the baseline. We use the end-to-end execution time of the application to compute the normalized performance. One can make the following observations. First, only using the In-PTE Directory Invalidation achieves an average of 27.3% performance improvement over the baseline, and only using

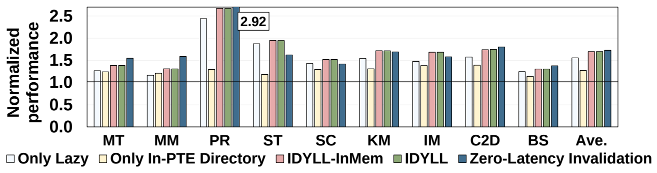

Figure 11: Performance of each scheme relative to baseline.

Lazy Invalidation achieves an average of 55.8%. With both optimizations enabled, IDYLL can provide an average improvement of 69.9%. The improvement brought by IDYLL is smaller compared to the sum of the improvements achieved by each optimization individually. This is because the two optimizations are complementary to each other and both reduce the contention with demand TLB miss requests. Second, the performance improvements of the two mechanisms on different applications vary. Many applications achieve higher performance improvement with Lazy Invalidation, as it reduces the latency and contention caused by all invalidation requests. In contrast, In-PTE Directory Invalidation only works for unnecessary invalidations. However, one can observe that MM benefits more from In-PTE Directory Invalidation. This is because the unnecessary invalidation requests thrash the IRMB in Lazy Invalidation and removing these unnecessary invalidations is immensely helpful. Finally, with both mechanisms enabled, the performance improvement is significant for high MPKI applications (shown in Table 3). For example, PR achieves 2.67× over the baseline. In contrast, the performance improvement of BS is moderate. However, for MT, which has the highest MPKI value while not achieving an expected performance improvement. This is because the percentage of invalidation requests in MT is much lower as shown in Figure 5. As such, invalidation requests have less impact on its performance. We also find that IM has a relatively lower MPKI value and a small percentage of invalidation requests, but achieves high performance improvement. This is because IM has a memory-intensive process of converting each patch of image data into a column, where handling page table walk latency can hardly be hidden by the computation context switching (e.g., warp scheduling) in GPUs. Reducing invalidation latency can significantly benefit the execution.

Figure 11 also shows the performance improvement of zero-latency invalidation normalized to baseline (the last bar). Our approach achieves a comparable performance improvement against zero-latency invalidation. Interestingly, we find that the performance improvements for some applications in our approach (e.g., ST, SC, and IM) are higher than the performance improvements of zero-latency invalidation. The reasons are twofold. First, our In-PTE Directory Invalidation reduces unnecessary invalidation requests being sent to GPUs, whereas in zero-latency invalidation, all invalidation requests are still sent to all GPUs. Therefore, our approach reduces interconnect congestion. Second, the IRMB in our design also serves as an indicator of invalid PTE. Thus, when a demand TLB miss request hits in the IRMB, we can bypass the local page table walk and directly send the request to the host (since the requested PTE would have become invalid in the local page table had we not delayed the invalidation). This reduces the page table walk required by the demand TLB miss compared to the zero-latency implementation, thereby reducing overall demand TLB miss request latency.

We also evaluate the performance of IDYLL-InMem (discussed in Section 6.4) in Figure 11. The IDYLL-InMem achieves a similar performance improvement (average 70%) as IDYLL (i.e., In-PTE Directory Invalidation design). The reason is twofold. First, both designs employ Lazy Invalidation which effectively reduces the contention caused by invalidation requests. Second, VM-Cache lookup and VM-Table lookup have much less latency compared to the host-side page table walk and can be performed in parallel with the host-side page table walk without incurring any additional latency compared to the In-PTE Directory Invalidation design. Furthermore, we observe an average hit rate of 60.2% of the VM-Cache. We do not observe memory bandwidth contention caused by VM-Table accesses when missing in the VM-Cache. Given the similar performances of both designs and the lack of space, our following experimental results are obtained from IDYLL (i.e., In-PTE Directory Invalidation design).

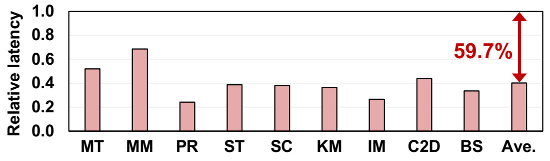

Figure 12: Demand TLB miss request latency.

Demand TLB miss request latency: To understand the reasons behind the performance improvements, we plot the demand TLB miss latency and page migration waiting latency when using IDYLL. Figure 12 shows the total latency of demand TLB miss requests in IDYLL normalized to the total latency of demand TLB miss requests in the baseline execution (the lower the better). As observed, the total demand TLB miss request latency of our approach is reduced by about 60% compared to the baseline. The reductions in demand TLB miss request latency directly translate to performance improvements. For example, for PR and IM, the total demand TLB miss request latency of our approach is only about 25% of the baseline, thus achieving a significant performance improvement.

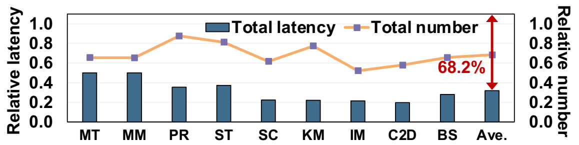

Figure 13: Total number of invalidation requests and total latency of invalidation requests.

Invalidation requests: The bar in Figure 13 shows the total latency of invalidation requests in IDYLL normalized to the total latency of invalidation requests in the baseline execution, and the line in the figure shows the percentage of invalidation requests in IDYLL normalized to the baseline. First, our approach eliminates all unnecessary requests so that the average number of invalidation requests in IDYLL is reduced by 32% of the baseline. The total invalid request latency in our approach is reduced by 68.2% compared to the baseline. This is because (i) the total number of invalidation

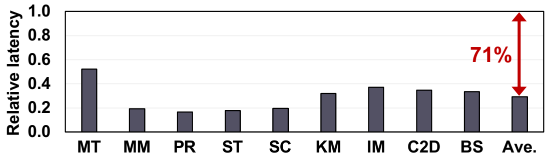

Figure 14: Page migration waiting latency.

requests is reduced and (ii) we coalesce multiple invalidation requests for page table walks, which can share the same page walk cache entries, thereby improving the L2 level page walk cache hit rate of invalidation requests and reducing the latency caused by page walk cache misses.

Page migration waiting latency: Figure 14 shows the total latency of page migration waiting in IDYLL normalized to the total latency of page migration waiting in the baseline execution. The results indicate an average of 71% latency reduction brought by our approach compared to the baseline. Recall that, in the baseline, page migration cannot start before (i) all the GPUs finish page table walks for the invalidation requests, and (ii) the host finishes page table walks for the invalidation. While the GPU page table walk can happen concurrently with the host page table walk, the walking latency on the host side is expected to be much lower compared to the GPUs. This is because of the high bandwidth of the host page table walk and the fewer far faults that need to be handled by the host page table walk. As a result, our approach significantly reduces the page migration waiting latency as our approach only needs to perform the host-side page table walk to determine the GPUs with valid translations and register them in the IRMB of the corresponding GPU without performing page table walks in the GPU. We also want to emphasize that our approach does not affect the page data migration time as our focus is address translation invalidations. The figure only shows page migration waiting time which does not include the page data migrating time.

7.2 Sensitive Study

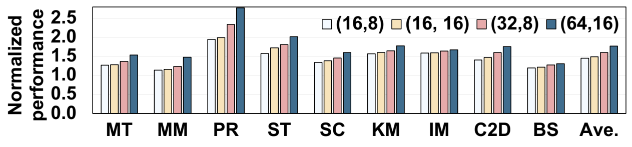

Figure 15: IDYLL with different IRMB size. The (x,y) on the legend indicates (size of bases, size of offsets).

IRMB size: Recall that, in our configuration of IDYLL, the IRMB is set to 32 different bases with 16 offsets per base. In this study, we evaluate the performance impact with different IRMB configurations. The (x,y) indicates (size of bases, size of offsets). As shown in Figure 15, first, the performance improvement decreases as the IRMB size reduces. When IRMB size is reduced to (16, 8), the average performance improvement is 44.8% over the baseline, which is 25.1% lower than the default sizes (i.e., 32 bases and 16 offsets). This is because fewer invalidation requests can be kept with a small-sized IRMB, which leads to frequent IRMB eviction, introducing more contention to demand TLB miss requests. Second, when increasing the IRMB size to (64, 16), the average performance improvement is 76.9%, which is 7% higher than the default size. Considering the hardware overhead, we choose (32, 16) as our configuration.

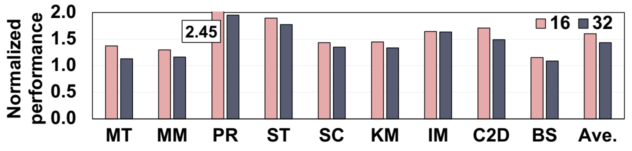

Figure 16: IDYLL with 16- and 32-threaded page table walk.

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

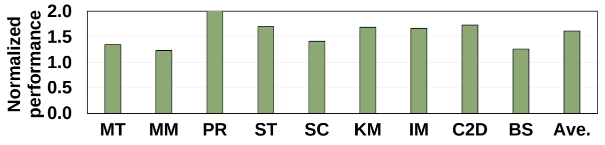

Figure 17: IDYLL with 2048-entry L2 TLB.

L2 TLB sizes: We evaluate IDYLL under a larger GPU L2 TLB size (2048 entries, 64-way, and 32-set). Figure 17 shows that IDYLL achieves 61.4% performance improvement over the baseline using 2048-entry L2 TLB. Although a larger TLB size can keep more translations into the TLB and reduce the number of page table walks in the GMMU, the TLB shootdown caused by page migration makes a large TLB much less helpful.

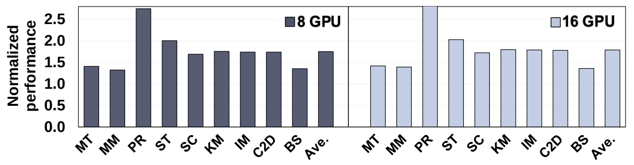

Figure 18: IDYLL with 8 and 16 GPUs.

Number of GPUs: We evaluate IDYLL in 8-GPU and 16-GPU systems. Figure 18 plots the performances of IDYLL with 8 GPUs and 16 GPUs normalized to the baselines with 8 GPUs and 16 GPUs. The average performance improvement of 8-GPU and 16-GPU is 75.3% and 79.1%, respectively. One can make the following observations. First, the performance improvement increases with more GPUs. Note that, for a fair comparison, we only increase the number of GPUs without changing the application's input dataset sizes. As a result, with more GPUs, the pages are more frequently shared across GPUs, and more page migrations are triggered, introducing more invalidation requests to each GPU. Our approach is effective in handling these significant invalidations by batch processing and lazy updates. Second, the trend of performance gains becomes slow as the number of GPUs increases. This is because, in our design, we map several GPU access bits to a single bit of the host-side PTE, which increases the false positive rate when sending invalidation requests to GPUs. However, the Lazy Invalidation mechanism still eliminates significant invalidation request interference to demand TLB miss requests. In a nutshell, IDYLL delivers performance improvements with more GPUs.

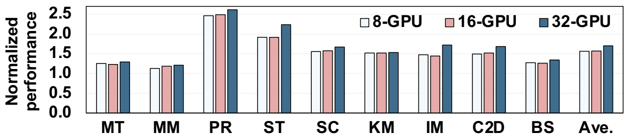

Figure 19: IDYLL with 4 unused bits and varying GPU count.

Number of unused bits: We next evaluate IDYLL with fewer unused bits (i.e., 4 unused bits). Figure 19 shows the performance of IDYLL under 8, 16, and 32 GPUs, using 4 unused bits. The results are normalized to the baseline execution with 8, 16, and 32 GPUs, respectively. Although the increased hash false positives when employing less number of unused bits can result in more unnecessary invalidation requests being sent to the GPU and potentially degrade the benefits one can obtain from In-PTE Directory Invalidation, our approach still achieves an average performance improvement of 56.5%, 57.1%, and 70.1%, respectively. This is mainly due to the effectiveness of Lazy Invalidation, which plays a significant role in enhancing the IDYLL performance.

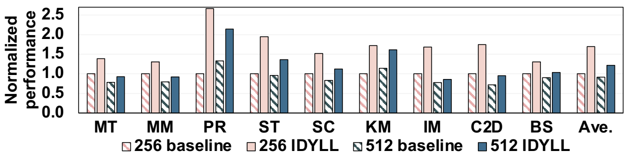

Figure 20: IDYLL with 512 access counter threshold.

Access counter threshold: In our discussion so far, we use the UVM default access counter threshold (256). We next evaluate the impact of a larger access counter threshold on IDYLL. Figure 20 shows the performances of baseline and IDYLL with a threshold of 512 (baseline-512 and IDYLL-512); the results are all normalized to the original baseline (baseline-256). One can make the following observations. First, IDYLL-512 outperforms baseline-512 by 30.0%. Second, this performance improvement is less than the improvement with a threshold of 256 (i.e., IDYLL-256 outperforms baseline-256 by 69.9%). This is because a higher threshold reduces the total number of page migrations, hence reducing the number of invalidations, making the potential for improvement less in our approach. However, we want to emphasize that having a larger access counter threshold does not necessarily guarantee better overall performance. Figure 20 also presents the baseline-512 performance normalized to the baseline-256. One can observe that the performance drops by 10% when using a threshold of 512. This is because a larger threshold leads to an increased number of remote accesses, exacerbating the NUMA overheads and causing a degradation in overall performance.

7.3 Adopting Large-sized Pages

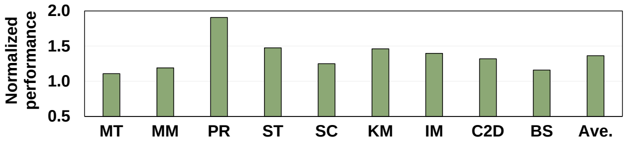

Figure 21: IDYLL with 2MB pages.

In this study, we evaluate how IDYLL works with a 2MB large page. To sufficiently stress the virtual memory subsystem with 2MB pages, we enlarge the input sizes for each application. Figure 21 shows the performance improvement of IDYLL with 2MB page size normalized to the baseline execution with 2MB page size. The results indicate that IDYLL achieves an average of 36.3% performance improvement. It is expected that the gains drop from 4KB page size as the large page size increases TLB reach, page walk cache hit rate, and reduces the page walk contention. However, our approach remains effective, especially for those applications with substantial page sharing (such as PR). This is because adopting a large page size increases false sharing and still incurs a sizable amount of invalidation requests in the system.

7.4 Compared to Page Replication

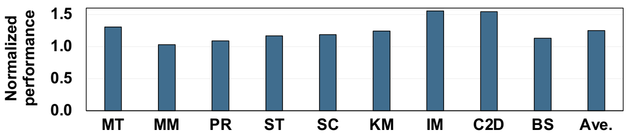

Figure 22: IDYLL with page replication.

We next compare IDYLLwith page replication. Different from access counter-based remote mapping, page replication allows the page to be duplicated among GPUs so read accesses do not require page migration. It consumes more physical memory space but avoids NUMA overheads as in access counter-based remote mapping. Figure 22 shows the performance of IDYLL normalized to the page replication. IDYLL achieves an average of 25.0% performance improvement. Comparing the results with the performance of access counter-based page migration in Figure 11, the improvement is less, especially for PR, ST, and SC. This is because these applications are read-intensive applications and the invalidation requests are significantly reduced, which makes less room for optimization. However, when a GPU performs a write/modification operation, the page replication approach coalesces all replications into a single page and each GPU needs to perform a page table walk to invalidate the corresponding PTE. Therefore, for write-intensive applications such as IM and C2D, our approach still significantly outperforms page replication. Note that, we do not simulate the oversubscription in this comparison experiment. Thus, with substantial sharing among multiple GPUs, page replication is not scalable and may decrease the overall performance.

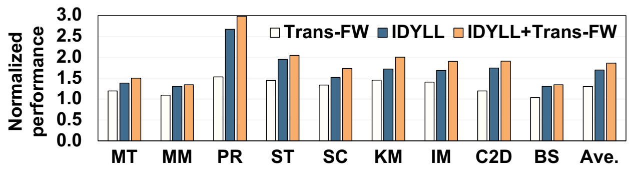

Figure 23: Comparison to Trans-FW [40].

7.5 Compared to State-of-the-art

We compare IDYLL with the state-of-the-art *Trans-FW* [40]. The Trans-FW expedites the far faults by leveraging the valid translation information residing in remote GPU's page tables. It implements false-positive hardware structures to perform remote lookups. For a fair comparison, we reduce the hardware overhead of Trans-FW to 720 bytes with 443 fingerprints in PRT (original 813 bytes with 500 fingerprints) to match our IRMB design hardware overhead. All other configuration parameters are kept the same as in Trans-FW. Figure 23 shows the performance of Trans-FW, IDYLL, and Trans-FW+IDYLL normalized to the baseline execution. One can make the following observations. First, Trans-FW achieves an average of 30% performance improvement over the baseline, and IDYLL outperforms Trans-FW by an average of 39.9% (69.9% – 30%). This is because Trans-FW does not optimize the invalidation requests, which is one of the major performance factors in multi-GPU executions. Second, our approach can be combined with Trans-FW to further improve the address translation efficiency. Specifically, IDYLL+Trans-FW achieves an average of 86.3% performance improvement over the baseline. This is because most unnecessary page table walks in the GMMU are reduced when combined IDYLL and Trans-FW, and the contention of page table walk threads in GMMU is further mitigated. Note that, Trans-FW and IDYLL are not completely orthogonal, this is because in our approach, we also bypass a fraction of unnecessary page table walks when requests hit IRMB.

7.6 DNN workloads

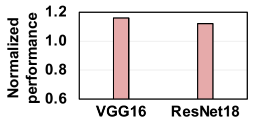

Figure 24: IDYLL with DNN workloads.

We also evaluate IDYLL with real DNN workloads, VGG16 and ResNet18. The layers of the DNN workloads are parallelized across multiple GPUs [39]. We use Tiny-Imagenet-200 [36], which contains 100,000 images of 200 classes, and each class consists of 500 training images, 50 validation images, and 50 test images. We do not use large datasets as the simulation time is too long (over months) on large datasets. As shown in Figure 24, IDYLL achieves a performance improvement of 15.9% and 12.0% on VGG16 and ResNet18, respectively. As the computation of each layer requires the use of the weights stored on each GPU, such substantial weight sharing causes page migrations and PTE invalidations, bringing the potential for performance improvement to our approach. The results demonstrate that the proposed IDYLL works with real-world multi-GPU workloads.

8 RELATED WORK

TLB Optimizations: Substantial prior works have focused on TLB optimizations, including software TLB optimizations [5, 12, 19, 34, 35, 77], and hardware optimizations [7, 38, 59, 60, 66, 75, 78, 80]. Li et ai. [41] proposed least-TLB design which leverages translation sharing and spilling to reduce TLB redundancy and improve the performance of multi-application execution. Mazumdar et al. [45] proposed a storage-efficient dead block predictor in the last level TLB to improve TLB hit rate. Shahar et al. [64] implemented a software address translation mechanism to build memory-mapped files on GPUs. Compared to TLB optimizations, our approach focuses on reducing the contention of page table walk resources caused by invalidation requests, which is orthogonal or complementary to the TLB optimizations.

Page Table Walk Optimizations: Techniques to accelerate the page table walk have been explored extensively, including optimizations of MMU cache, page migration, and prefetching [8, 29, 43, 44, 52, 55, 65]. Achermann et al. [1] proposed to reduce the NUMA effects of page walks by replicating and migrating page tables. Pratheek et al. [61] introduced a dynamic page walk stealing, which restricts uncontrolled interference in page walks for multi-tenancy. However, none of these efforts considered the performance impact of invalidation page table walks introduced by page migrations. In our work, we reveal the significant bottlenecks caused by invalidation requests. We propose a software-hardware co-design to effectively eliminate the performance burdens.

NUMA Optimizations: Non-Uniform Memory Access (NUMA) systems arise as computing becomes more and more heterogeneous across multiple processors within a single system [15, 17]. In order to optimize NUMA systems, prior works [10, 13, 22, 32, 46, 76, 81] have focused on improving the performance of multi-GPU scenarios. Besides, Multi-Chip Module (MCM) designs have emerged to cope with the growing demand for computing capabilities with small chiplets. Arunkumar et al. [6] proposed MCM-GPU, which takes advantage of preserving locality by allocating threads close to the data. Pratheek et al. [62] proposed MCM-aware GPU virtual memory to improve the performance over shared TLB. These works focused on page migration optimizations to reduce remote data access, while no one has looked at the address translation issues caused by page migration.

9 CONCLUSION

This paper analyzes the impact of page table invalidations on the performance of address translation in multi-GPU systems. Our investigation reveals that page table invalidation requests significantly increase the demand TLB miss request latency and page migration waiting latency. To address this problem, we propose IDYLL that employs an In-PTE Directory Invalidation mechanism to reduce unnecessary invalidations sent to the GPU, and a Lazy Invalidation that batches invalidations and lazy updates them to the GPU local page table. Experimental results show that the IDYLL improves performance by 69.9% on average.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous MICRO reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2011146, #2154973, #1725657, #1910413, and #2312157.

REFERENCES

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. 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*. 283–300.

[2] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 141–150. https://doi.org/10.1109/IPDPS49936.2021.00023

[3] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[4] AMD. 2015. *AMD APP SDK OpenCL Optimization Guide*.

[5] 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

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

[7] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 136–150.

[8] 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* (Saint-Malo, France) *(ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

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

[10] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A Mojumder, José L Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 596–609. https://doi.org/10.1109/HPCA47549.2020.00055

[11] 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* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/3410463.3414639

[12] 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* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[13] Leul Belayneh, Haojie Ye, Kuan-Yu Chen, David Blaauw, Trevor Mudge, Ronald Dreslinski, and Nishil Talati. 2022. Locality-aware Optimizations for Improving Remote Memory Latency in Multi-GPU Systems. In *2022 31th International Conference on Parallel Architectures and Compilation Techniques*. https://doi.org/10.1145/1122445.1122456

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

[15] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2017. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00795ED1V01Y201708CAC042

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

[17] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE micro* 30, 2 (2010), 16–29.

[18] NVIDIA Corp. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[19] 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* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[20] Yue Dai, Youtao Zhang, and Xulong Tang. 2023. CEGMA: Coordinated Elastic Graph Matching Acceleration for Graph Matching Networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 584–597. https://doi.org/10.1109/HPCA56546.2023.10070956

[21] 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* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702

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

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

[24] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2022. Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems. *arXiv preprint arXiv:2203.15981* (2022).

[25] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M LaConte. 2013. Medical image processing on the GPU–Past, present and future. *Medical image analysis* 17, 8 (2013), 1073–1094.

[26] Siying Feng, Subhankar Pal, Yichen Yang, and Ronald G Dreslinski. 2019. Parallelism analysis of prominent desktop applications: An 18-year perspective. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 202–211.

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

[28] 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* (Munich, Germany). ACM, New York, NY, USA, 413–423. https://doi.org/10.1145/2591635.2667189

[29] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1109–1124. https://doi.org/10.1145/3373376.3378494

[30] 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

[31] 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* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[32] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G Rogers. 2020. Locality-centric data and threadblock management for massive GPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1022–1036.

[33] Ian King. 2017. *Chipmakers Nvidia, AMD Ride Cryptocurrency Wave—for Now*. www.bloomberg.com/news/articles/2017-07-17/chipmakers-nvidia-amd-ride-cryptocurrency-wave-for-now.

[34] 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* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 651–664. https://doi.org/10.1145/3173162.3173198

[35] 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* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[36] Ya Le and Xuan Yang. 2015. *Tiny imagenet visual recognition challenge*. http://cs231n.stanford.edu/

[37] Jiwon Lee, Ju Min Lee, Yunho Oh, William J Song, and Won Woo Ro. 2023. SnakeByte: A TLB Design with Adaptive and Recursive Page Merging in GPUs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1195–1207.

[38] Bingyao Li, Yueqi Wang, and Xulong Tang. 2023. Orchestrated Scheduling and Partitioning for Improved Address Translation in GPUs. In *The 60th ACM/IEEE Design Automation Conference (DAC)*.

[39] Bingyao Li, Qi Xue, Geng Yuan, Sheng Li, Xiaolong Ma, Yanzhi Wang, and Xulong Tang. 2022. Optimizing Data Layout for Training Deep Neural Networks. In *Companion Proceedings of the Web Conference 2022*. 548–554.

[40] Bingyao Li, Jieming Yin, Anup Holey, Youtao Zhang, Jun Yang, and Xulong Tang. 2023. Trans-FW: Short Circuiting Page Table Walk in Multi-GPU Systems via

Remote Forwarding. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 456–470. https://doi.org/10.1109/HPCA56546.2023.10071054

[41] 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-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1154–1168.

[42] Sheng Li, Geng Yuan, Yue Dai, Youtao Zhang, Yanzhi Wang, and Xulong Tang. 2022. SmartFRZ: An Efficient Training Framework using Attention-Based Layer Freezing. In *The Eleventh International Conference on Learning Representations*.

[43] Zhulin Ma, Yujuan Tan, Hong Jiang, Zhichao Yan, Duo Liu, Xianzhang Chen, Qingfeng Zhuge, Edwin Hsing-Mean Sha, and Chengliang Wang. 2020. 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, 255–262.

[44] 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* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[45] Chandrashis Mazumdar, Prachatos Mitra, and Arkaprava Basu. 2021. Dead page and dead block predictors: Cleaning tlbs and caches together. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 507–519.

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

[47] Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. 2021. GPS: A Global Publish-Subscribe Model for Multi-GPU Memory Management. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 46–58.

[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. 2018. *NVIDIA DGX-2*. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-2/dgx-2-print-datasheet-738070-nvidia-a4-web-uk.pdf

[50] NVIDIA. 2022. *NVIDIA Linux Open GPU Kernel Module Source*. https://github.com/NVIDIA/open-gpu-kernel-modules

[51] NVIDIA Corp. 2018. *EVERYTHING YOU NEED TO KNOW ABOUT UNIFIED MEMORY*. https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[52] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. 2021. 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*. 194–210.

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

[54] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk 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

[55] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every walk's a hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 128–141.

[56] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel 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

[57] 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* (Vancouver, B.C., CANADA) *(MICRO-45)*. IEEE Computer Society, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[58] Binh Pham, Ján Veselý, Gabriel H Loh, and Abhishek 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

[59] 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* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

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

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

[62] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2022. Designing Virtual Memory System of MCM GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 404–422.

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

[64] Sagi Shahar, Shai Bergman, and Mark 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

[65] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 180–192. https://doi.org/10.1109/ISCA.2018.00025

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

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

[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* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230

[69] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David 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, 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* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 191–204. https://doi.org/10.1145/3410463.3414633

[71] Tech Power Up. 2017. *ETH Mining: Lower VRAM GPUs to be Rendered Unprofitable in Time*. www.techpowerup.com/234482/eth-mining-lower-vram-gpus-to-be-rendered-unprofitable-in-time

[72] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and Norman 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

[73] Tyler Allen. 2023. *UVM performance evaluation*. https://github.com/tallendev/uvm-eval

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

[75] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek 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

[76] 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* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3307650.3322247

[77] 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* (Phoenix, Arizona) *(ISCA '19)*. ACM, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223

[78] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek 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

[79] Juheon Yi and Youngki Lee. 2020. Heimdall: mobile GPU coordination platform for augmented reality applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–14.

[80] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. 2018. Filtering Translation Bandwidth with Virtual Caching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 113–127. https://doi.org/10.1145/3173162.3173195

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

[82] Shulin Zhao, Haibo Zhang, Cyan Subhra Mishra, Sandeepa Bhuyan, Ziyu Ying, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita Das. 2021. HoloAR: On-the-fly Optimization of 3D Holographic Processing for Augmented Reality. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 494–506.