STAR: Sub-Entry Sharing-Aware TLB for Multi-Instance GPU

Bingyao Li
University of Pittsburgh
bil35@pitt.edu

Yueqi Wang
University of Pittsburgh
yuw249@pitt.edu

Tianyu Wang
University of Pittsburgh
tiw81@pitt.edu

Lieven Eeckhout
Ghent University
lieven.eeckhout@ugent.be

Jun Yang
University of Pittsburgh
juy9@pitt.edu

Aamer Jaleel
NVIDIA
ajaleel@nvidia.com

Xulong Tang
University of Pittsburgh
tax6@pitt.edu

***Abstract*—NVIDIA's Multi-Instance GPU (MIG) technology enables partitioning GPU computing power and memory into separate hardware instances, providing complete isolation including compute resources, caches, and memory. However, prior work identifies that MIG does not partition the last-level TLB (i.e., L3 TLB), which remains shared among all instances. To enhance TLB reach, NVIDIA GPUs reorganized the TLB structure with 16 sub-entries in each L3 TLB entry that have a one-to-one mapping to the address translations for 16 pages of size 64 KB located within the same 1 MB aligned range. Our comprehensive investigation of address translation efficiency in MIG identifies two main issues caused by L3 TLB sharing interference: (i) it results in performance degradation for co-running applications, and (ii) TLB sub-entries are not fully utilized before eviction. Based on this observation, we propose STAR to improve the utilization of TLB sub-entries through dynamic sharing of TLB entries across multiple base addresses. STAR evaluates TLB entries based on their sub-entry utilization to optimize address translation storage, dynamically adjusting between a shared and non-shared state to cater to current demand. We show that STAR improves overall performance by an average of 28.7% across various multi-tenant workloads.**

***Index Terms*—multi-instance GPU, sub-entry TLB**

I. Introduction

Graphics Processing Units (GPUs) are extensively utilized in contemporary computing systems to accelerate performance across various applications. As artificial intelligence/ML models evolve, GPU manufacturers are continually enhancing the capabilities of individual GPUs to meet the surging computational demands [16], [20], [26], [52], [61], [64]. However, previous research has shown that these emerging applications still cannot fully exploit the existing GPU computational resources due to different workloads facing various resource bottlenecks and exhibiting different sensitivities to different resources [6], [25], [29], [38], [58], [62].

To address the issue of underutilization, GPU vendors are evolving to offer GPU resource partitioning capabilities to enable multiple applications to share the same physical GPU resources. NVIDIA's Multi-Instance GPU (MIG) [36] is one of the prominent GPU-sharing technologies. MIG enables a single physical GPU to be divided into several isolated instances, each with its own set of resources, including streaming multiprocessors (SMs), local memory, and caches. NVIDIA MIG is designed to offer isolation of resources for each instance, ensuring performance without interference from other instances. However, a recent study [63] has indicated that while MIG effectively partitions most of the memory system, it does not partition the last-level TLB (i.e., L3 TLB). The shared L3 TLB allows the TLB to dynamically allocate its entries based on the demand from various instances, optimizing the use of the available TLB capacity. On the flip side, TLB sharing also leads to contention among multi-tenant applications.

With the increasingly large data sets and wide memory footprints of applications, the TLB has become a critical performance bottleneck [32], [33], [41], [42], [44], [50]. Expanding the TLB size to alleviate this issue is impractical due to hardware size constraints. In response, NVIDIA's new generation GPU (e.g., A100) presents an innovative TLB architecture to enhance TLB reach. Specifically, in the L2 and L3 TLBs, an entry comprises 16 sub-entries, each directly corresponding to the address translation of 16 sequential 64 KB pages within a contiguous 1 MB-aligned segment, as recently revealed through reverse-engineering [63]. By compressing multiple translations into a single TLB entry, the TLB can manage more data with fewer entries, thereby reducing hardware overhead, while improving TLB efficiency and boosting overall performance. A sub-entry TLB design performs well for isolated workloads that use large contiguous memory, however, in multi-tenant setups where the L3 TLB is shared, this design can lead to sub-entry underutilization because interference from co-runners causes frequent evictions when only a portion of the sub-entries are used.

To understand the impact of L3 TLB contention in a multi-tenant environment, we co-run representative GPU applications on an NVIDIA A100 GPU with MIG enabled, see Figure 1 where each application runs within its own instance while sharing the L3 TLB. The GPU is partitioned into varying sizes of instances, including (3g, 2g, 2g) and (3g, 3g), where 'g' represents the allocation of computing resources; each instance runs a single application. Performance is normalized to each application running alone on its respective instance, thereby having exclusive access to the L3 TLB. We observe

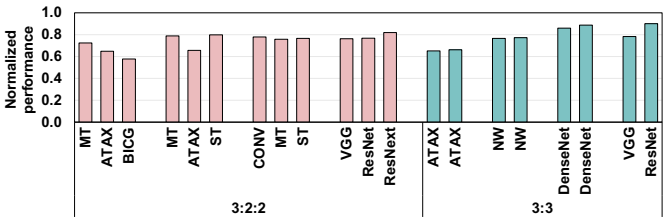

Fig. 1. Performance of co-running applications on NVIDIA's A100.

that L3 TLB contention significantly degrades the performance of individual applications. This is because high access demand and interference from co-running applications lead to severe TLB thrashing. This thrashing extends the reuse distance of address translations, making translations less likely to be reused before they are evicted. It also leads to lower sub-entry utilization at the point of eviction, as the interference from concurrent requests accelerates TLB eviction (quantitative results and detailed analysis are given in Section IV).

A large body of prior work focused on improving address translation efficiency from multiple perspectives, including contiguity-based range TLBs [24], [28], [41], [56], cluster TLBs [42], [44], employing large pages [8], [40], [43], TLB compression [51], and TLB speculation [10]. Many of these optimizations are designed for single GPU/CPU setups running one application and are not effectively applicable to MIG environments with multiple tenants. First, range-TLB, cluster-TLB, and TLB-compression strategies are optimized for sequential and stride memory access patterns, commonly found within individual applications. Co-running applications often have varied and unpredictable access patterns, making it challenging for these TLB optimizations to consistently capture the requested translations efficiently. Second, TLB speculation relies on the assumption of consistent access patterns to achieve accurate predictions. Similarly, in scenarios where the L3 TLB is accessed by multiple applications simultaneously, the interference between applications disrupts the regularity of memory accesses, significantly diminishing prediction accuracy. Third, using large pages can increase TLB reach by reducing the number of TLB entries needed to cover the same memory range. However, multi-tenant environments often host a mix of applications with diverse memory access patterns. While some applications can efficiently leverage large pages, others with irregular or sparse access patterns may not observe the same benefits. This variance leaves those less suited to large pages still facing contention issues. Other work, for example MASK [9], improves address translation efficiency in multi-application environments by controlling warp access to the shared TLB through an epoch- and token-based scheme. Although this approach is effective at reducing TLB thrashing, it helps little with TLB sub-entry utilization.

Motivated by these challenges, we systematically investigate and optimize the address translation in MIG systems. Our quantitative analysis reveals that contention in the L3 TLB critically undermines MIG performance, primarily due to low utilization of TLB sub-entries caused by multi-tenant interference. To address this, we propose the **S**ub-En**T**ry Sh**A**ring-Awa**R**e (STAR) TLB, which dynamically allows different base addresses to share TLB entries. Specifically, instead of defaulting to Least Recently Used (LRU) eviction upon receiving a new address translation, our method evaluates and selects an entry based on its current sub-entry utilization that satisfies the sharing criteria for inserting the new address translation. Additionally, our approach can dynamically switch between a TLB entry's shared and non-shared states, adapting to the fluctuating demands of TLB sub-entries. We make the following contributions:

- We show that a major performance bottleneck in MIG arises from severe contention in the shared L3 TLB. We provide a detailed analysis of how multi-tenant interference affects address translation reuse and TLB sub-entry utilization.
- We propose STAR, a hardware design tailored to mitigate the negative effects of multi-tenant interference and enhance overall application performance. STAR enables multiple base addresses to share the same TLB entry, enhancing sub-entry utilization. It also dynamically switches between shared and non-shared states to adapt to varying application demands.
- We show that STAR improves overall performance by an average of 28.7% across a suite of multi-tenant workloads. We show that STAR outperforms various TLB design alternatives and is orthogonal to these approaches to achieve further performance improvement.

II. Background

A. Multi-Instance GPU

Modern GPUs, such as NVIDIA's Ampere and Hopper generations (e.g., A100 and H100), leverage Multi-Instance GPU (MIG) technology to enhance resource utilization by enabling the sharing and partitioning of GPU resources [1], [37]. MIG technology allows a single GPU to be divided into multiple GPU partitions, each operating as an independent GPU instance with its own dedicated resources. The partitioning includes SMs and the entire memory system, including the on-chip crossbar ports, L2 cache banks, memory controllers, and DRAM address buses, effectively eliminating performance interference between different applications. Each GPU instance contains at least one GPU Processing Cluster (GPC) along with a designated portion of the GPU's memory. The current setup of MIG can support up to seven distinct instances, offering predefined configurations including 1g, 2g, 3g, 4g, and 7g, where 'g' indicates a portion of the total GPU compute resources. For instance, the smallest configuration available is 1g.5gb, providing 1/7 of the Streaming Multiprocessors (SMs) and 5 GB of GPU memory. However, configurations for 5g and 6g are not available.

The TLB organization in MIG is shown in Figure 2. Specifically, MIG partitions the L1 and L2 TLBs along with the GPCs: the L1 TLB is shared between the two SMs within each Texture Processing Cluster (TPC), and the L2 TLB is shared across the SMs of a GPC. However, interestingly, prior work [63] reveals that the L3 TLB in today's GPUs

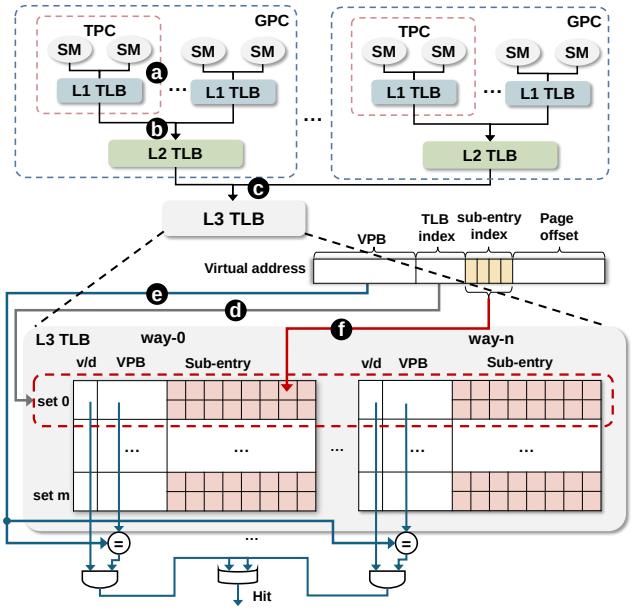

Fig. 2. TLB structure and address translation process in A100.

(e.g., NVIDIA's Ampere generations) remains shared across all instances in MIG-supported GPUs. This sharing indicates that despite MIG's comprehensive approach to partitioning, the last-level TLB still lacks the isolation necessary to prevent contention across different GPU instances.

B. Address Translation in MIG

Figure 2 also illustrates the address translation process in MIG. Upon a memory request, the L1 cache is first checked. The L1 TLB lookups are performed upon an L1 cache miss (a). If the request misses in the L1 TLB, it first checks the L1 Miss Status Holding Register (MSHR) to coalesce the same requests, and the outstanding request is sent to the L2 TLB for lookup (b). Similarly, requests missing in the L2 TLB are sent to the L3 TLB (c), and requests that miss the L3 TLB are further sent to the GPU memory management unit (GMMU) to perform page table walks. If the page table walk fails, a local page fault is generated and propagated to the host CPU to resolve. It then initiates the target data transfer and updates TLBs, caches, and page tables. Finally, the address translation request is replayed after resolving the page fault.

TLB sub-entries: Traditionally, each TLB entry would directly map one virtual page to one physical page. This is a straightforward, one-to-one relationship: each entry in the TLB represents a single page of memory, as typically done in L1 TLBs in the latest NVIDIA GPUs (e.g., NVIDIA's Ampere generations). However, these GPUs organize their L2 and L3 TLB entries differently to increase TLB reach [63]. Specifically, each of these entries contains 16 sub-entries, which directly map to the address translations for 16 pages. These pages can be either 64 KB or 2 MB in size, and all of them fall within an aligned range of either 1 MB or 32 MB in size, respectively. That means each sub-entry in a TLB entry has a one-to-one relationship with a single page. Note that, in the sub-entry setting, if any TLB entry is evicted, all the 16 sub-entries associated with that TLB entry are invalidated.

TABLE I
BASELINE MULTI-INSTANCE GPU CONFIGURATION.

Module	Configuration
SM	1 GHz, 108 in total
DRAM	5 GB per slice
L1 D-cache	64 KB, 2-way set associative
L1 I-cache	32 KB, 2-way set associative
L2 cache	2 MB per slice, 8-way set associative
L1 TLB	16 entries, 16-way, 1-cycle lookup latency, TPC shared, LRU replacement policy
L2 TLB	128 entries, 8-way, 16 sub-entries per entry, 10-cycle lookup latency, GPC shared, LRU replacement policy
L3 TLB	1024 entries, 8-way, 16 sub-entries per entry, 40-cycle lookup latency, GPU shared, LRU replacement policy
Page table walk	8 page table walkers, GPC shared, 100-cycle latency per level [46], [47], [54]
Page walk cache	128 entries shared across page table walkers [46]

Address translation in a sub-entry TLB proceeds as follows. The virtual address of memory access is partitioned into a virtual page number (VPN) and a page offset. The lower bits of the VPN are further divided into a TLB index and a sub-entry index, and the higher bits of the VPN serve as the virtual page base (VPB). During a TLB lookup, the TLB index is first used to identify the corresponding set (d). Then, the VPB from the virtual address is compared with the VPB within the set to check for a hit or miss (e). If there is an entry hit, the process further checks the sub-entry index to determine if the specific sub-entry is present in the TLB entry (f). A non-zero sub-entry indicates a TLB hit. Conversely, a zero sub-entry or no matching VPB results in a TLB miss, triggering a page table walk. When a valid translation for a virtual address is found and if this virtual address is within the range covered by an existing TLB entry, the translation is added to the appropriate sub-entry slot. If no existing TLB entry covers this address range, a new entry is created. This involves evicting the least recently used (LRU) entry along with zeroing all of its 16 sub-entries. The new translation is then inserted into the corresponding sub-entry slot of the newly established TLB entry.

III. METHODOLOGY

A. Baseline Configuration

We use MGPUSim [49] throughout the paper. To model multi-instance GPU, we substantially modified MGPUSim by adding (i) different cache, memory, and SM configurations for different instance sizes, (ii) a shared L3 TLB (with sub-entries) and sub-entries for the L2 TLB, and (iii) GMMUs for each instance, including page walk cache, page walk queue, page table walk thread, and the page table. Note that the exact latency of a page walk depends on whether it hits the page walk cache and whether it needs to wait for an available page walk thread in the page walk queue. These processes are all faithfully modeled in the simulator. In this paper, we focus on a GPU partitioned into instances of sizes 3g, 2g,

TABLE II
LIST OF APPLICATIONS.

Abbr.	Application	Benchmark Suite	Instruction Count	L2 TLB MPKI	Access Pattern
ATAX	Matrix Transpose and Vector Multiplication	Polybench	328,441,844	204.7	Stream, Stride
BICG	Sub Kernel of BiCG-Stab Linear Solver	Polybench	321,758,896	208.9	Stream, Stride
FFT	Fast Fourier Transform	SHOC	409,534,464	0.5	Stream, Stride
BFS	Breadth First Search	SHOC	94,727,760	5.6	Random
ST	Stencil 2D	SHOC	59,289,600	21.9	Stream, Block
FIR	Finite Impulse Resp.	Hetero-Mark	192,675,840	0.3	Stream
PR	Page Rank	Hetero-Mark	316,669,952	0.44	Random
MT	Matrix Transpose	AMDAPPSDK	9,564,256	205.0	Stride
NW	Needleman–Wunsch	Rodinia	87,909,120	38.4	Stream, Dependent
CONV	Convolution 2D	DNN-Mark	2,629,570,744	1.9	Stream, Stride

and 2g. The number of SMs, cache size, and memory size are partitioned proportionally based on each instance size. Each instance runs a single application. Our approach also applies to various combinations of instance sizes and we provide a sensitivity study with altering instance sizes in Section VI-B. The baseline configuration is listed in Table I. The page size is set to 64 KB as the MIG default configuration.

B. Applications

We use 10 applications from the Polybench [45], SHOC [17], Hetero-Mark [48], AMDAPPSDK [7], Rodinia [15], and DNN Mark [18] benchmark suites, which are representative real-world applications. The details of the applications are listed in Table II. These applications have different computation intensities. For example, FFT and CONV are compute-intensive and heavily use floating-point operations, whereas BICG and ATAX involve memory-intensive operations. The applications also cover a wide range of access patterns. Specifically, in the stream access pattern, data is accessed sequentially, offering good locality and predictability. In contrast, the stride access pattern, shown in operations like matrix transpose, involves accessing data at a constant stride, leading to non-sequential memory accesses. For example, in MT, accessing elements column-wise in a row-major stored matrix or vice versa involves memory accesses with a stride equal to the number of rows or columns. In the dependent access pattern, certain data is accessed depending on the computation results of previous elements, such as in NW, where each cell's computation in the scoring matrix depends on the values of its neighboring cells. In the block access pattern, data is accessed in blocks or chunks. For example, in ST, data is divided into blocks that fit into the cache, allowing for efficient computation of the convolution operation over each block. In the random access pattern, data is accessed in an irregular and unpredictable order, commonly observed in graph traversal algorithms (e.g., BFS) and web-ranking algorithms (e.g., PR).

To study multi-instance execution, we use the applications listed in Table II to form multi-application workloads. We also include applications with their smaller input size (indicated as ApplicationName_s) to balance the application execution times within the workload. Table III shows the eleven workloads, each consisting of three applications. The workloads are

TABLE III
MULTI-TENANCY WORKLOADS.

Abbr.	Workload	Applications	Category
W1	workload1	MT, ATAX, BICG	HHH
W2	workload2	MT, ATAX, ST	HHM
W3	workload3	MT, NW, ST	HMM
W4	workload4	MT_s, ST_s, FIR	HML
W5	workload5	MT_s, BFS, PR	HML
W6	workload6	MT_s, FFT, FIR	HLL
W7	workload7	NW, CONV, ST_s	MMM
W8	workload8	ST_s, NW, FFT	MML
W9	workload9	BFS, BFS, PR	MML
W10	workload9	ST_s, FIR, FFT	MLL
W11	workload10	FFT, FFT, FIR	LLL

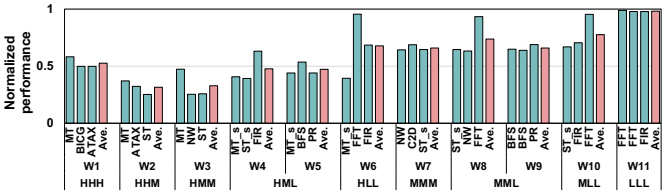

Fig. 3. The normalized performance of each application within the workload.

formed by analyzing the L3 TLB access intensity of each application. Specifically, we measure each application's misses-per-kilo-instructions (MPKI) of the address translations at L2 TLB. Applications are then grouped into three categories based on their L2 TLB MPKI values: Low (L, MPKI<1), Medium (M, 1<MPKI<100), and High (H, MPKI>100). Accordingly, workloads are formed representing various combinations of these categories, including HHH, HHM, HMM, HML, HLL, MMM, MML, MLL, and LLL. Given the possibility of some applications finishing earlier than others during simultaneous execution, we adopt the same strategies as previous studies to ensure continuous TLB contention [33], [46], [59]. That is, applications that are completed early are re-run until the completion of the longest-running application within the workload. The statistical data is gathered only during the initial complete run of each application within any given workload.

IV. QUANTITATIVE ANALYSIS OF MIG MULTI-TENANCY

A. Overall Performance Characteristics

In a MIG-enabled GPU, the shared L3 TLB is a key source of contention under multi-tenant execution. To quantify the performance impact of interference and contention at the L3 TLB, we study normalized performance of individual applications within workloads and the average performance of the nine workloads in Table III as shown in Figure 3. Specifically, the normalized performance here is the performance of an application executed in conjunction with other applications, normalized to the performance of running alone. The average performance is calculated as the harmonic mean of normalized performance for all applications within a workload. Note that, when an application runs alone, it uses the same instance size but gets exclusive use of the full L3 TLB capacity. One can make the following observations. First, L3 TLB contention compromises the performance of individual applications. In

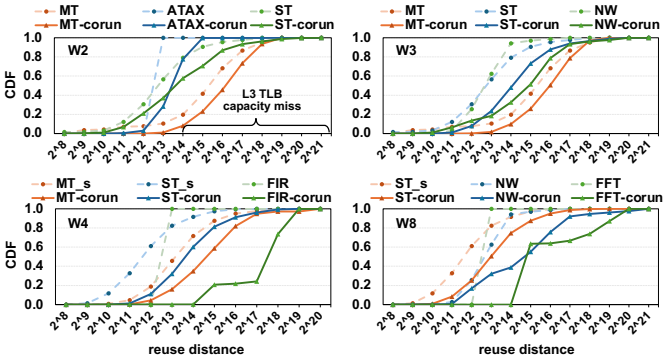

Fig. 4. CDF of translation reuse distances at the L3 TLB.

W11, each application experiences a negligible performance overhead. Conversely, in W1, there is an average performance drop of 48%. Overall, the performance decreases by an average of 40% compared to applications run independently. Second, the performance degradation varies among different applications within the same workload. This variance is particularly significant in applications with higher MPKI values. As shown in W8, where the performance of FFT, with a low MPKI of 0.5, drops by 6.6%, in contrast to NW, which suffers a substantial 36.7% decrease with a medium MPKI of 38.4. This is because applications with higher MPKI values are more sensitive to TLB misses due to limited latency hiding through context switching or other parallel threads. Third, the performance degradation of the same application can vary depending on the specific co-runners. Taking ST_s as an example, its performance drops by 61% in W4 but only by 34% in W10. This is due to the co-running applications having a higher MPKI in W4 than those in W10, which leads to more severe L3 TLB contention.

We further investigate the reuse distance of translations for multi-tenancy. The reuse distance is defined as the unique translation count between two accesses to the same translation from the same instance. The unique translations include all distinct translations originating from either the same application or different applications. Specifically, we measure the reuse distance of address translations that reach the L3 TLB.

Figure 4 presents the Cumulative Distribution Function (CDF) of the translation reuse distances of four workloads with representative MPKI mix, i.e., HHM, HMM, HML, and MML. For comparison, we also show the reuse distance for each application running alone, depicted in light dotted lines. We observe that some applications (e.g., NW, FFT and FIR) show very different reuse distances when they execute concurrently with others versus running alone. For example, in its single-run of NW, 94.2% translation reuses are less than the L3 TLB capacity (i.e., 16,384 sub-entries), indicating a higher possibility of these reuses being accommodated within the TLB. However, in W3, only 32.7% of the reuses in NW are within the TLB capacity. This is because in W3, ST and MT have high/medium MPKIs, and they generate a large number of translation requests to the L3 TLB. Therefore, the reuse distance of NW is extended. For applications such as ST_s in

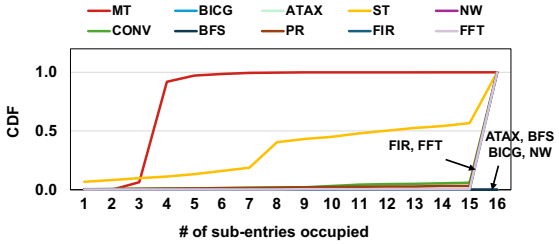

Fig. 5. CDF of TLB sub-entry utilization when running in isolation.

W8, its reuse distance shows relatively little change compared to its isolated run. This is because it generates intensive translation requests that miss in the L2 TLB and therefore consume a considerable portion of L3 TLB entries; at the same time, its co-located application FFT has a lower MPKI, thereby generating less contention for TLB resources. We also marked the L3 TLB capacity in the figure. It is observed that for applications with severe contention (e.g., MT), more than 80% of the translation reuses miss in the L3 TLB.

B. Sub-Entry Utilization Characterization

Recall that, when a TLB entry is evicted, the sub-entries within the TLB entry are also evicted. This design is beneficial for scenarios where memory accesses exhibit a contiguous or linear pattern. In such cases, most sub-entries can be utilized effectively before any TLB entry is evicted, thereby maximizing the efficiency of the TLB. However, in situations where memory access patterns are non-contiguous, particularly in workloads with sparse or irregular memory access patterns, some sub-entries might remain unused at the time of eviction. We therefore study the utilization of sub-entries of each application when it is evicted. Figure 5 shows the CDF of sub-entry utilization of all applications listed in Table II running in isolation. One can observe that applications with streaming access patterns, such as FIR and FFT, tend to make full use of TLB sub-entries before eviction due to their sequential access nature. In contrast, the application MT exhibits low sub-entry utilization (most TLB entries evicted with only four sub-entry occupied). This is because MT has stride access patterns, where accesses do not align well with the contiguous page mappings of the sub-entries. Moreover, application ST, which exhibits a block access pattern along with a stream pattern, shows nearly 50% of the TLB entries are evicted when only half of the sub-entries are utilized. This is because of the mixed nature of its memory accesses, i.e., sequential within blocks but non-contiguous between them. Note that the memory footprints of applications ATAX, BFS, BICG, and NW can fit in the address coverage range of L3 TLB, therefore no eviction is observed when they are running alone.

We further analyze the contention and interference impact on sub-entry utilization when co-running applications. Figure 6 presents the sub-entry utilization of six workloads with representative MPKI mix, i.e., HHH, HHM, HMM, HML, MMM, and LLL. The darker solid lines in the figure represent the sub-entry utilization of each application when co-running; we also show the sub-entry utilization when applications run

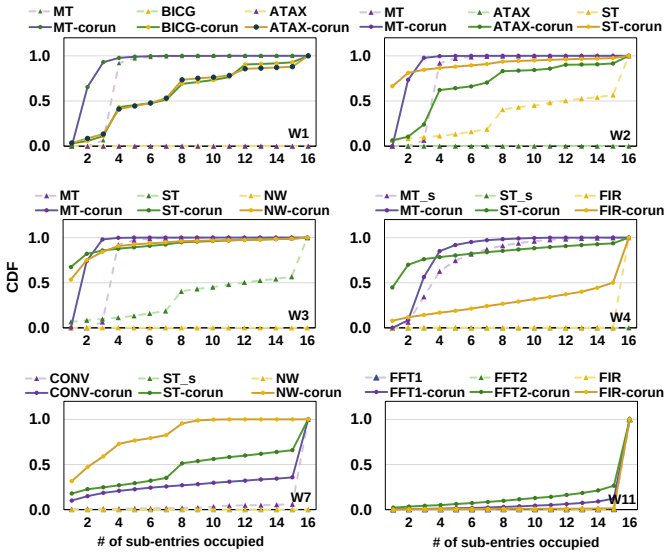

Fig. 6. CDF of TLB sub-entry utilization under co-running.

individually in lighter dot lines. One can make the following observations. First, all applications within workload categories, except LLL, show substantially less utilization of sub-entries when a TLB entry is evicted compared to applications that are run in isolation. For example, for ATAX in W1, 73.4% of its TLB entries are evicted when less than half of the sub-entries are used, even though there are no evictions when run in isolation. Similarly, for application ST in W2, 66.3% of its TLB entries are evicted with only one sub-entry used, whereas 43.4% of its TLB entries are evicted with fully occupied sub-entries when run in isolation. More severe underutilization is observed for workloads with a larger MPKI mix. Second, an application may incur very different utilization in different workloads. For example, ST_s in W4 has 69.8% of its TLB entries with just two sub-entries used at the time of eviction. In contrast, in W7, ST_s shows only 22.5% of its entries evicted with two sub-entries used. This is because in W4, the co-running application MT_s has high MPKI, which leads to a greater number of translation requests to the L3 TLB, causing ST_s to suffer from more frequent evictions before the sub-entry is fully utilized due to increased contention.

Does promoting sub-entry to regular TLB entry solve the problem? A straightforward approach to enhancing sub-entry utilization would be to convert sub-entries into regular TLB entries, thus eliminating the 1 MB virtual address range alignment for each TLB entry and allowing any address to utilize these sub-entries. However, such an expansion would result in a significant hardware cost. In the baseline, each way uses one comparator to match the incoming address with stored tags. Requests that fall within a specific TLB entry range directly map to the corresponding sub-entry, simplifying comparisons. However, allowing any address to use a sub-entry would require each of the 16 sub-entries in a TLB entry to have its comparator. Since each TLB entry is associated with 16 sub-entries, this would increase the number of comparators by 16 for each way. We use CACTI [53] to estimate the TLB size: under this design, the TLB size is 17.2× of the baseline. This increase is impractical considering the constraints on GPU die size. Therefore, it is important to explore a more efficient and cost-effective approach to optimize TLB sub-entry utilization without excessively increasing its size.

V. Sub-Entry Sharing-Aware TLB

Our goal in this paper is to improve the MIG-enabled GPU TLB hit rate, thereby boosting the performance of multi-tenancy execution. While contention for a shared resource is inevitable in environments where resources are limited, our analysis in the previous section has revealed opportunities to mitigate contention effects by optimizing the utilization of TLB sub-entries.

To this end, we propose STAR, a hardware-supported TLB sub-entry sharing mechanism that allows multiple base addresses to share a TLB entry of 16 sub-entries dynamically. Our approach organizes sub-entries into multiple groups, allocating each group to one base address for usage. However, implementing an effective and efficient dynamic sub-entry sharing mechanism is non-trivial and faces several challenges. First, reducing the number of sub-entries allocated to each base address changes the direct mapping from the original design. It is important to resolve any resulting conflicts while maintaining the correctness of address translation lookup. Second, it is important to select appropriate base addresses for sharing and determine when to share such that the utilization can be maximized and minimize the performance impact compared to the original sub-entry capacity. Third, enabling sub-entry sharing, the TLB lookup and insertion procedure should not significantly be increased compared to the baseline. Finally, the proposed TLB sub-entry sharing should involve minimum hardware overheads, offering a cost-effective and scalable alternative to merely enlarging the TLB size.

A. Sub-Entry Sharing-Aware TLB Format

Figure 7 depicts the format of virtual addresses and the content of a sharing-aware TLB entry. Specifically, the original 4-bit sub-entry index is split into an n-bit sub-entry index and a $(4-n)$-bit Address Identify Bit (AIB). The value of n depends on how many base addresses are sharing one TLB entry. In the current design, we allow each original TLB entry to support two base addresses, and each address can occupy eight sub-entries (i.e., $n = 3$). This pre-determined value is based on our characterization analysis presented in Section IV-B, where we found over half of the TLB entries were evicted while less than half of their 16 sub-entries were utilized. The shared TLB entry also needs additional bits to maintain the metadata (e.g., valid/dirty bits) for each base address separately. Since each base address is limited to using 8 sub-entries in our design, the absence of a direct one-to-one mapping within a 1 MB alignment could lead to conflicts for sub-entries with identical index bits. To address this, our approach dictates that if a sub-entry is already in use and a new request arrives with the same sub-entry index bit, the new request will replace the existing

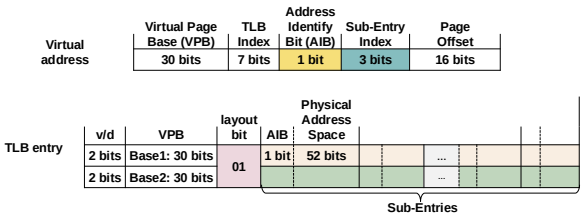

Fig. 7. Format of the virtual address and contents of a sharing-aware TLB entry in sequential layout.

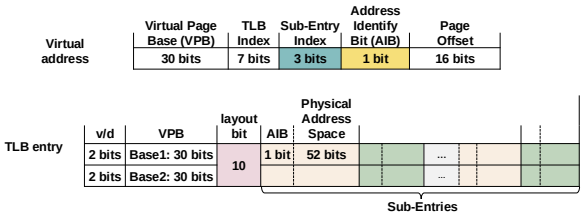

Fig. 8. Format of the virtual address and contents of a sharing-aware TLB entry in stride layout.

one. Consequently, at any given moment when a TLB entry is shared, only one address translation with a particular sub-entry index bit can be present in a TLB entry. The Address Identify Bit (AIB) becomes essential here, serving to identify which address is currently using the sub-entry.

Because of the diverse access patterns exhibited by applications, e.g., stream versus stride patterns, we introduce a flexible method that dynamically allocates sub-entries to base addresses based on the usage patterns of sub-entries. Specifically, for scenarios where sub-entries are occupied sequentially, we allocate the first half of the sub-entries to the first base address and the second half to the second base address. In this case, the last three bits of the sub-entry index are used to identify positions within each base's allocated sub-entries. The first bit of the sub-entry index acts as the Address Identify Bit (AIB) (shown in Figure 7). Alternatively, if the occupied sub-entries show stride access patterns, the sub-entries are interleavedly allocated between the two base addresses according to the stride size. In our approach, we pre-define stride size to 1. That is, the first base address is assigned to sub-entries with even indices, whereas the second base address is allocated to those with odd indices. In this case, the first three bits of the sub-entry index are used to determine the location of sub-entries (shown in Figure 8). These sub-entry layout strategies are recorded in layout-bit (initially set to '00', indicating non-shared status). When inserting the new base address, the choice between sequential or stride layout depends on the *current occupancy pattern* of sub-entries: a consecutive pattern triggers the sequential layout (layout bit set to '01'), whereas a non-consecutive pattern activates stride layout (layout bit set to '10'). The layout bit then determines which index bits are used during lookup.

Note that choosing between different numbers of shared base addresses leads to a trade-off between sub-entry utilization and hardware overhead. More shared base addresses increase sub-entry utilization but require more bits to be stored in the TLBs and more cycles to compare each base address. On the other hand, fewer shared base addresses reduce hardware overhead and lookup latency, but lead to lower sub-entry utilization. We provide sensitivity results with different numbers of shared base addresses in Section VI-B.

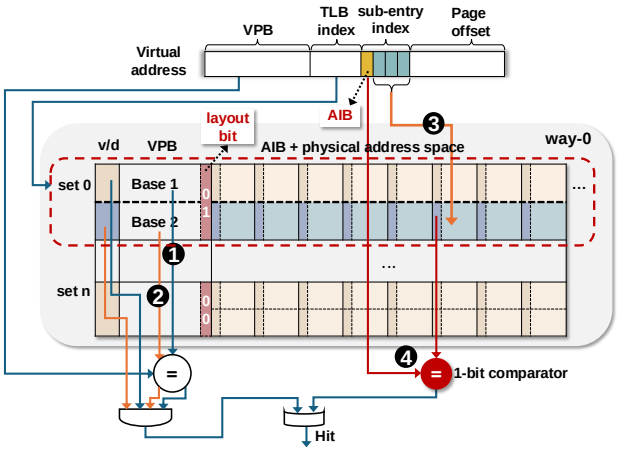

Fig. 9. TLB lookup process in STAR.

B. TLB Lookup and Insertion Process

When to share? In our sub-entry sharing-aware TLB architecture, sub-entry sharing is allowed under specific conditions to optimize utilization. Initially, the TLB works as the default baseline, with each TLB entry independently managed until all entries (ways) within the TLB set are allocated. At that point when a new address arrives, instead of proceeding with a Least Recently Used (LRU) entry eviction, we first check how many sub-entries are actually being used in all entries of that set. An entry is considered eligible for sharing if it meets the following criteria: (i) less than eight sub-entries are utilized, and (ii) only one base address is currently occupying the entry. If multiple entries fit these criteria, we prefer to pair the incoming base address with an existing entry from the same process because access patterns within the same process tend to be similar. If no matching process is found, we choose the candidate where the current sub-entry utilization is the lowest. Only if no entries meet these conditions for sharing do we fall back to the baseline approach of evicting the least recently used entry and inserting the new one.

STAR also supports dynamic shifts between the shared and non-shared state. The shared TLB entry can still revert to the non-shared state. Specifically, when a TLB entry, currently shared between two base addresses, reaches a state where all 8 sub-entries allocated to one base are fully utilized, it indicates the potential for an increase in demand for this process. Upon the arrival of a new request that cannot be accommodated due to fully utilized sub-entries for its corresponding base address, the shared TLB entry will be reverted to being exclusively used by one base with increasing demand. The other base and its associated sub-entries are evicted from the TLB entry. The TLB entry status is updated, which involves resetting the layout bit, the metadata for the second base, and reorganizing the sub-entries based on the 4-bit sub-entry index.

Algorithm 1: TLB Lookup with Sub-Entry Sharing.

```
/* Lookup () */
Request arrives L3_TLB
Compare each entry in set with request's VPB in parallel
if non-shared TLB entry then
    Use 4-bit of sub-entry index
    if find matched sub-entry then
        TLB hit, respond with PFN concatenated with page offset
    else
        TLB miss, send request to GMMU for page table walk
else
    Check base addresses sequentially
    if find matched base address then
        Check layout bit
        if layout bit equals '01' then
            Use last three bits of sub-entry index
        else
            Use first three bits of sub-entry index
        Locate sub-entry and compare AIB with request's AIB
        if AIB matches then
            TLB hit
        else
            TLB miss
    else
        TLB miss
```

Algorithm 2: TLB Insertion with Sub-Entry Sharing.

```
/* Insert ()*/
Find the TLB set for the virtual address.
/* Scenario 1: Base address hit                    */
if address matches an existing base then
    if entry is shared then
        Use the layout bit to find the sub-entry
        if base's sub-entries are full then
            Make entry non-shared
            Reset the layout bit
            Reorganize sub-entries
        else
            Insert into sub-entry. Evict the original if needed
    else
        Insert translation with 4-bit index
else
    /* Scenario 2: Miss all base addresses         */
    if there is an available entry in the set then
        Insert new base address into first vacant entry
    else
        if conditions for sub-entry sharing are met then
            Check access pattern of sub-entries
            if sub-entries are continuously occupied then
                Apply sequential layout; set layout bit to '01'
            else
                Apply stride layout; set layout bit to '10'
            if sub-entry is already occupied by the original base then
                Try to relocate the original entry
                if alternative sub-entry is also occupied then
                    Evict the original entry to accommodate new
                    translation
            else
                Insert new address translation with determined layout
        else
            Evict least recently used (LRU) entry and insert new address
```

TLB Lookup: Figure 9 shows the TLB lookup procedure, which is provided in Algorithm 1. Specifically, when a translation request arrives at the L3 TLB, it first identifies the corresponding set. Each entry in that set is compared in parallel with the request's VPB. Two scenarios can happen. First, the entry is non-shared with only one base address, indicated by the layout bit set to '00'. The 4-bit sub-entry index is then used to locate the corresponding sub-entry. Second, if an entry contains two base addresses, these addresses are checked sequentially (❶, ❷). Upon identifying a matching entry, the layout bit is checked to determine which bits should be used to index the corresponding sub-entry (❸). If the layout bit is set to '01', the last three bits of the sub-entry index are employed to locate the specific sub-entry. On the other hand, if the layout bit is set to '10', the first three bits will be used. Once locating the sub-entry, the Address Identify Bit (AIB) stored in the sub-entry is compared with the request's AIB (❹). A matching AIB indicates a TLB hit, and the PFN stored in the sub-entry is concatenated with the page offset to form the requested physical address. An AIB miss (also TLB miss) is handled the same way as in the baseline which involves sending the request to the GMMU for a page table walk. In our sequential L3 TLB lookup, the lookup latency for a single base address remains the same as the baseline at 40 cycles. This latency includes the cycles required for the request to reach the L3 TLB, base address comparison, data access, and TLB miss resolution. Given that comparing base address typically requires 5-10 cycles [22], [23], we conservatively add an additional 10 cycles for the second base address comparison. Thus, if both base address lookups are required, the total lookup latency for this TLB entry amounts to 50 cycles. All these latency overheads are included in our evaluation.

TLB Insertion: The insertion algorithm is shown in Algorithm 2. When a new address needs to be inserted, the index bits of the virtual page number are used to determine the set in the TLB. Two scenarios can happen depending on whether the address matches an existing base address in the identified set. In the first scenario, if the base address hits, the following process depends on the shared status of the TLB entry. For a non-shared TLB entry (layout bit '00'), the address translation is inserted into its corresponding sub-entry using the complete 4-bit sub-entry index. On the other hand, if the TLB entry is shared by two base addresses, the layout bit determines whether the last or first three bits of the index are used to locate the sub-entry. In a situation where all sub-entries for the inserted base address are full, it triggers a shift from a shared to a non-shared state. That is, the metadata and sub-entries associated with the other shared base address are evicted and the layout bit is reset to '00'. The sub-entries will be relocated using the 4-bit sub-entry index. Instead, if the sub-entries of the inserted base address are not full, the incoming translation is inserted into the sub-entry as indicated by the 3-bit sub-entry index; if the target location is already occupied, the original translation is removed.

In the second scenario, it misses all base addresses. If there is an available entry within the set, the new base address is inserted into this first vacant entry. Otherwise, the conditions for sub-entry sharing are evaluated, as previously discussed. If an entry is selected for sharing with the new base, the access pattern of the current entry is checked to determine

how to organize the shared sub-entries (i.e., sub-entry layout). Specifically, if the sub-entries are occupied continuously without any gaps, it is classified as a sequential pattern. For such cases, the sequential layout will be applied to the sub-entries, and the layout bit is set to '10' (indicating a sequential layout). The last three bits of the sub-entry index are used to assign the address translation to its corresponding sub-entry. In contrast, if there are empty slots among these sub-entries, the pattern is identified as stride. The stride layout is then employed (the layout bit is set to '10'), utilizing the first three bits of the sub-entry index to map the address translation to a sub-entry. It is important to note that when allocating a new address translation to a sub-entry, it may happen that the sub-entry is already in use by the original base address. In such cases, we will attempt to relocate the original entry to another sub-entry sharing the same index bits if it is unoccupied. If this alternative sub-entry is also in use, the original address translation that initially occupied the chosen sub-entry will be evicted to accommodate the incoming new address translation. Note that the insertion into the L3 TLB is off the critical path and hence does not directly impact performance.

C. Hardware Overhead

In our configuration, the L3 TLB entries are augmented with additional bits to support the new functionality: a layout bit for sub-entry indexing layout, and an Address Identify Bit (AIB) to identify which address currently occupies the sub-entry. Each TLB entry now comprises two bases, with associated valid/dirty bits, the virtual page base (VPB), AIB, and physical address space (PAS). Therefore, the sharing-aware TLB entry format requires an additional 2 bits (layout bit) + 16 bits (1 bit AIB per sub-entry) + 30 bits (second base address) + 2 bits (v/d for second base address) = 50 bits per TLB entry. Given that our L3 TLB design accommodates 1024 entries, and each entry originally consists of 864 bits (2-bit v/d, 30-bit VPB, and a 52-bit PAS per sub-entry), the introduction of sub-entry sharing and associated metadata increases the size per entry to 914 bits. Additionally, our design adds a 1-bit comparator for each sub-entry to match the AIB. We use CACTI [53] to estimate the area and power overhead of our approach. The result shows 1.4% area overhead over the original L3 TLB assuming a 22 nm technology node. Dynamic and leakage power consumption increase by 0.3% and 5.3%, respectively. Considering that TLBs account for a minor fraction of the total system dynamic power (less than 1% [34], [51]), the additional power overhead introduced by STAR is negligible to the overall system power consumption.

VI. Evaluation

A. Overall Performance

Figure 10 shows the performance improvements of individual applications within each workload and the harmonic average performance improvements (represented by the rightmost bar for each workload) of the multi-tenant workloads. Results are normalized to the baseline multi-tenant execution.

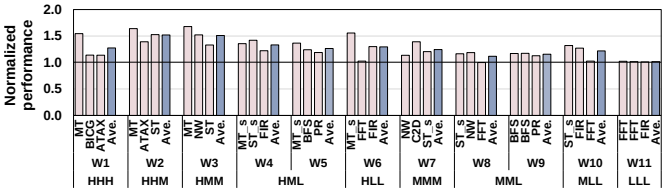

Fig. 10. Normalized performance improvements offered by STAR.

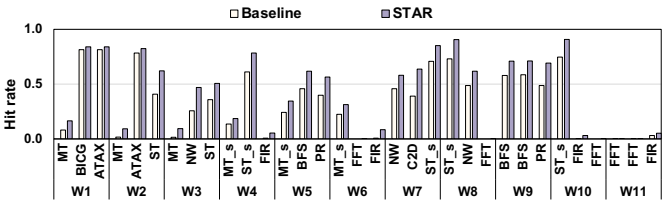

Fig. 11. L3 TLB hit rate of baseline and STAR.

Figure 11 plots the L3 TLB hit rate for each application in each workload. One can make the following observations. First, STAR improves performance by up to 51.3% with an average of 28.7% across all workloads. The improvement is more significant for workloads that suffer from severe contention in the L3 TLB (i.e., high MPKI value). For example, W2 (HHM) achieves 51.3% improvement and W7 (MMM) achieves 23.5% improvement, respectively. This is because workloads with high MPKI values benefit more from TLB optimizations as each TLB miss leads to costly page table walks, directly impacting performance. Our scheme effectively increases L3 TLB reach by sharing sub-entries, and as a result, the TLB can accommodate more translations and also capture a larger fraction of reused translations. Interestingly, `FIR` has very low MPKI (0.3) while having 27.1% performance improvement in W10 (MLL). This is because a large number of pending requests are coalesced to the same TLB miss in the L2 MSHRs. Reducing TLB miss latency significantly benefits the whole execution.

Second, the performance benefits come mainly from the enhanced L3 TLB hit rates through sub-entry sharing. On average, STAR improves L3 TLB hit rate by 32.8% across all workloads. For example, the L3 TLB hit rate of `ST` in W2 improves by 52%, which translates into a 52.7% performance improvement. The improved TLB hit rate also indicates an extended TLB reach, effectively reducing the number of (expensive) page table walks.

Third, the performance improvement of the same application differs across workloads. For example, `MT_s` achieves a substantial 55% performance improvement in W6 versus 35% in W4. This variance can be attributed to how the other applications within a workload interact with each other. In W6, the stride access patterns of the co-located application `FFT` result in low sub-entry utilization under the baseline scenario. When STAR is applied, `MT_s` is able to dynamically share the sub-entries that would otherwise remain underutilized by `FFT`. Since the latter application does not fully utilize its allocated sub-entries, sharing them with `MT_s` brings little to no detriment to its performance, hence the significant

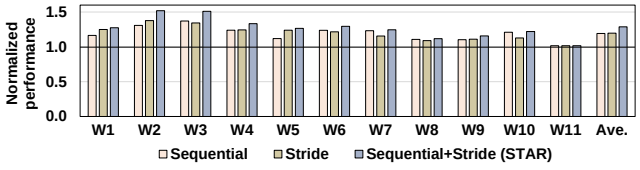

Fig. 12. Performance of different layout strategy in STAR.

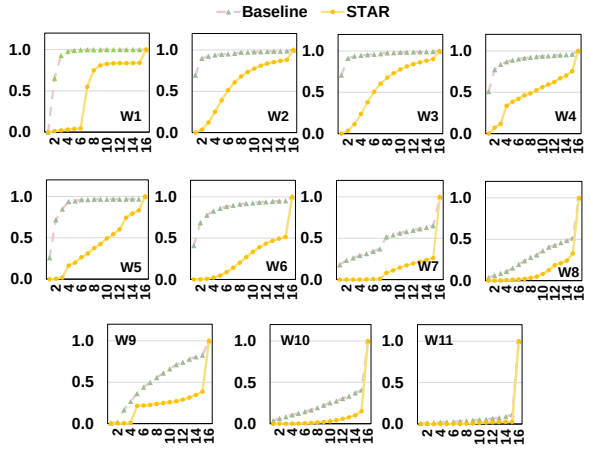

Fig. 13. CDF of sub-entry utilization of STAR.

improvement for MT_s. Conversely, in W4, applications such as ST_s and FIR make more efficient use of their allocated sub-entries, leaving fewer opportunities for MT_s to benefit from sharing.

Finally, our approach does not comprise the performance of any shared applications within the workload. This is because our approach can dynamically shift between the shared and non-shared states based on application demand. When an application has a higher demand for sub-entries, our approach allows for exclusive access to all sub-entries, as in the baseline scenario, thus maintaining performance integrity.

In Figure 12, we analyze the performance of different layout strategies, including sequential layout, stride layout, and a combination of both (i.e., STAR). The results show that sequential only, stride only, and sequential+stride achieve an average of 19.2%, 19.7%, and 28.7% performance improvement over the baseline, respectively. Workloads featuring a stride access pattern (e.g., W1) perform better with the stride-only layout. In contrast, workloads dominated by stream applications benefit more from the sequential layout (e.g., W10). Note that, the improvement brought by STAR is less than the sum of the improvements achieved by each layout individually. While combining the two layouts should theoretically capture the benefits of both, in practice, the mechanism that determines which layout to use may not accurately predict the most effective layout for the upcoming second base address. This is because the layout is determined by the first base address that occupies a sub-entry. If the first base address has a sequential access pattern, the second base is consequently forced to adopt the sequential layout, regardless of its optimal characteristics.

We further demonstrate the effectiveness of our approach by plotting sub-entry utilization, as shown in Figure 13. We observe that STAR consistently achieves higher utilization rates than the baseline, as indicated by the curves of STAR lying closer to the bottom right compared to the baseline, indicating a larger proportion of TLB entries with higher sub-entry utilization. We calculate the average utilization by summing up the product of the utilization fraction and the number of occurrences for each eviction and dividing by the total number of evictions. Our approach achieves on average 31.4% improvement in sub-entry utilization over the baseline.

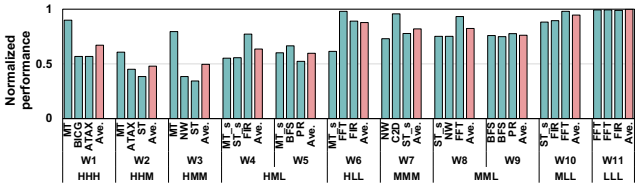

Fig. 14. Normalized performance for each application within its workload mix under STAR.

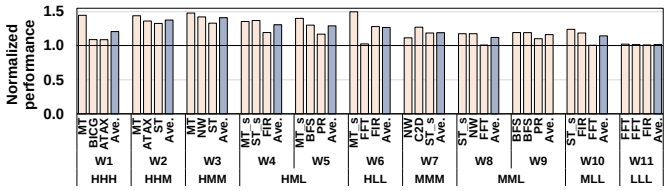

Fig. 15. STAR with a 4-base sharing TLB.

Figure 14 reports the performance for each application when run alongside others under STAR normalized to its performance when run in isolation. Per-application performance drops by an average of 26.1% under STAR. In contrast, under the baseline, performance is reduced by an average of 40% (as previously reported in Figure 3). This demonstrates that our approach effectively reduces the impact of multi-tenancy interference and improves overall performance.

B. Sensitivity Analyses

Different number of shared base addresses: In our discussion so far, up to two base addresses can share the same TLB entry. We now explore the option of having more base addresses sharing the same entry (i.e., up to 4 base addresses). That is, we allow scenarios where a TLB entry is used by one, two and four base addresses. The 4-base sharing mechanism works as follows. For entries with one or two base addresses, the process remains identical to our initial design. If an entry already has two base addresses and each utilizes fewer than four sub-entries, we enable sharing among four base addresses within that entry. To facilitate the varied sharing configurations (1, 2 or 4 base addresses), we introduce a 3-bit layout indicator. The initial state '000' indicates no sharing, with the last two bits specifying the sub-entry layout strategy. When the entry is shared, the last two bits deviate from '00' as in our initial design, and the first bit indicates the current number of shared base addresses in the entry. For example, '001' indicates two bases sharing with a sequential layout, while '110' represents four bases sharing with a stride layout. Our design also incorporates the ability to dynamically transition

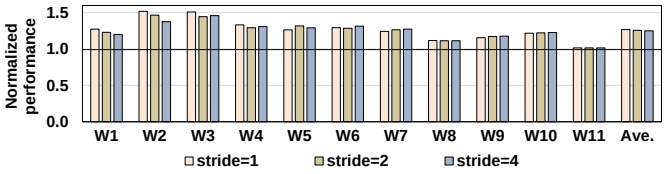

Fig. 16. STAR with different stride sizes.

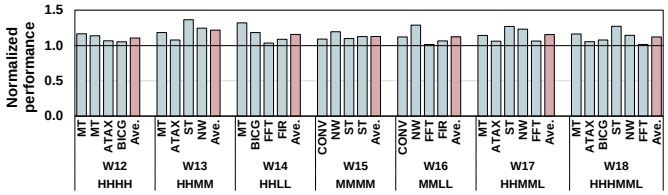

Fig. 17. STAR with different instance sizes.

between non-shared, 2-base address shared, and 4-base address shared states within the TLB. When a base address exhibits an increased demand for sub-entries, and if the entry is currently shared by four bases, we reduce the sharing to two bases, and further one base to accommodate the demand.

Figure 15 shows the performance of 4-base sharing normalized to baseline execution. On average, 4-base sharing improves performance by 22.7% over the baseline. However, it experiences a 6% performance reduction when compared to 2-base sharing. This is because, 4-base sharing, while enhancing utilization, introduces a trade-off by increasing address conflict evictions. Specifically, four addresses are allocated to share a single sub-entry (compared to two addresses sharing one sub-entry in the initial design), the address conflict evictions increase, which potentially reduces the TLB hit rate. Moreover, the lookup process requires up to four sequential operations in 4-base sharing, further exacerbating the lookup latency. We also evaluate the hardware overhead of the 4-base sharing approach with CACTI, and it shows a 3.6% area overhead compared to the baseline.

Different stride sizes: We initially configured our stride layout with a stride size of 1. Next, we study how varying stride values affect our approach. Given that two base addresses share the same entry, the feasible stride options are limited to 2, 4, and 8. Since a stride size of 8 corresponds to a sequential layout (refer to 'Sequential' bar in Figure 12), we focus on presenting the performance of our approach with stride sizes of 2 and 4, normalized to the baseline, as illustrated in Figure 16. It is important to note that while we vary the stride sizes of the stride layout, the dynamic layout selection and the shared to non-shared state switching mechanism remain unchanged. The results indicate that STAR with different stride sizes yields similar performance improvement. However, for workloads dominated by stride access patterns (e.g., W1 and W2), configurations with stride sizes of 2 and 4 perform worse than stride size of 1. This is because a stride size of 1 ensures that stride access patterns, whether they involve small or large strides, the translations are distributed across different sub-entries, effectively reducing conflicts. Conversely, while larger stride sizes may accommodate small stride access patterns efficiently, they tend to increase conflicts for larger stride access patterns. Note that, in W5, larger stride sizes outperform a stride size of 1 because the non-uniform distribution favored by stride sizes of 2 and 4 is better aligned for applications with irregular access patterns (i.e., `BFS` and `PR`).

Different instance sizes: We use the applications in Table II to form multi-tenant workloads with different numbers of applications, including five workloads with four applications each, one workload with five applications, and one with six applications, as listed in Table IV. The whole GPU is partitioned into different instance sizes depending on the number of co-running applications, with each instance running one application. Specifically, in W12-W16 (4-application workload), the GPU is divided into 2+2+2+1; in W17, it is divided into 2+2+1+1+1; and in W18, it is 2+1+1+1+1+1. Figure 17 reports normalized performance for STAR. First, our approach is able to deliver scalable performance improvements with different instance sizes, achieving 14.6%, 15.3%, and 12.1% performance improvement for the 4-, 5- and 6-application workloads, respectively. Second, the performance improvement is reduced as the number of co-running applications increases. This is because, first, the decrease in instance size leads to a corresponding reduction in L2 TLB size, which in turn increases the number of requests directed to the L3 TLB. Second, the increase in the number of co-running applications intensifies the competition for the limited number of L3 TLB entries which also impacts performance.

TABLE IV
MULTI-TENANCY WORKLOADS WITH 4, 5 AND 6 APPLICATIONS.

Abbr.	Workload	Applications	Category
W12	workload12	`MT, MT, ATAX, BICG`	HHHH
W13	workload13	`MT, ATAX, ST, NW`	HHMM
W14	workload14	`MT, BICG, FFT, FIR`	HHLL
W15	workload15	`CONV, NW, ST, ST`	MMMM
W16	workload16	`CONV, NW, FFT, FIR`	MMLL
W17	workload17	`MT, ATAX, ST, NW, FFT`	HHMML
W18	workload18	`MT, ATAX, BICG, ST, NW, FFT`	HHHMML

C. Comparison to TLB Alternatives

We compare STAR with three TLB design alternatives, which feature 8 sub-entries per TLB entry while doubling the number of ways or sets to keep total TLB capacity constant relative to the baseline. The TLB entries are exclusively used by one base address. Specifically, we consider (i) Half-Sub-Double-Set: 256 sets, 8 ways, and 8 sub-entries per entry; (ii) Half-Sub-Double-Way-Para: 128 sets, 16 ways, and 8 sub-entries per entry — here we increase the number of comparators with the number of ways such that all ways are compared in parallel within a set; this approach significantly increases hardware overheads; and (iii) Half-Sub-Double-Way-Seq: 128 sets, 16 ways, and 8 sub-entries per entry — we keep the same number of comparators as the baseline to avoid significant hardware overheads, such that two ways within a set are checked sequentially. We compare the hardware overhead of these alternatives using CACTI: both Half-Sub-Double-Set and Half-Sub-Double-Way-Seq maintain overheads comparable to

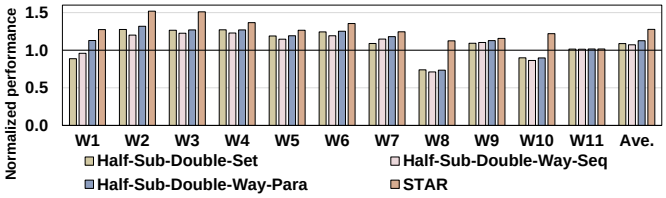

Fig. 18. Comparison of different TLB designs.

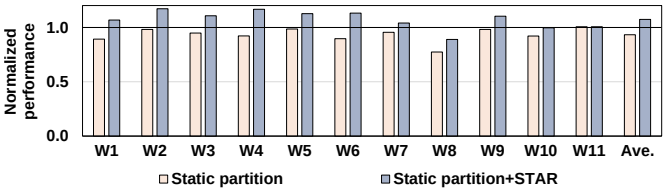

Fig. 20. Comparison to static partitioning.

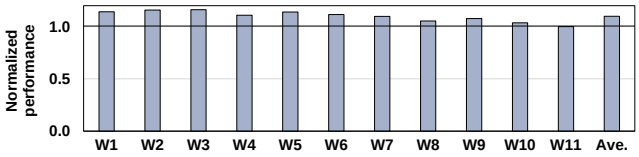

Fig. 19. STAR with a large 2 MB page.

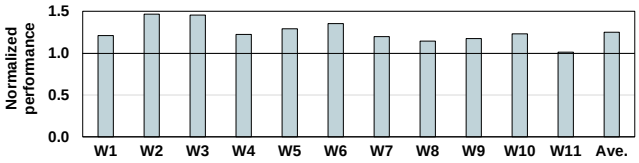

Fig. 21. Comparison with MASK [9].

the baseline with a 1.1% area increment; while Half-Sub-Double-Way-Para incurs a significant 78.8% area increment due to the increased number of comparators.

Figure 18 reports performance for each alternative compared to STAR. All results are normalized to the baseline multi-tenant execution. One can make the following observations. First, STAR achieves the highest performance improvement among all alternatives. Specifically, our approach achieves a 20%, 21.5%, and 16.1% performance improvement over Half-Sub-Double-Set, Half-Sub-Double-Way-Seq, and Half-Sub-Double-Way-Para, respectively. Second, halving the number of sub-entries statically to 8 incurs a performance degradation. For example, in W8, the performance of Half-Sub-Double-Way-Para drops by 26.7% compared to the baseline. This is because, in the baseline TLB design with 16 sub-entries, a hit in any of the 16 sub-entries reduces the chance of the TLB entry being evicted by the LRU scheme, potentially keeping it in the TLB longer, benefiting other accesses to one of the 16 sub-entries. This especially benefits the access patterns with good spatial locality, where multiple accesses are closely within a contiguous memory (e.g., `ST`). In contrast, reducing the number of sub-entries to 8 weakens the capability to exploit spatial locality (i.e., each hit now only benefits 8 sub-entries, compared to 16 in the baseline). Our approach dynamically alternates between a shared and non-shared TLB state, effectively enhancing the TLB sub-entry utilization while maintaining the efficiency of spatial locality accesses.

D. Comparison to Large Pages

We now evaluate how STAR performs with a 2 MB large page compared to a baseline with a 2 MB large page size, see Figure 19. STAR achieves an average of 10% performance improvement. This demonstrates that STAR remains effective when adopting a larger page size. The performance improvement is less compared to the 64 KB page size results. This is because large pages naturally enhance TLB reach and reduce the contention for limited TLB capacity. Nevertheless, large pages still suffer from sub-entry underutilization, and the eviction of a TLB entry covering a large address space can have a more severe impact on performance due to the broader range of addresses affected. Therefore, our approach continues to be effective as it enhances TLB sub-entry utilization, improves the hit rate, and increases overall performance.

E. Comparison to Static TLB Partitioning

One straightforward solution to mitigate contention is to statically partition the L3 TLB. In this approach, we statically partition the L3 TLB ways based on instance sizes. Specifically, in our setup, the instances are sized as 3g, 2g, and 2g. Accordingly, the TLB ways are allocated as one instance (the largest, at 3g) is assigned four ways of the TLB, while the other two instances (both 2g) are allocated two ways each. Figure 20 plots the performance of the static partition normalized to the baseline shared L3 TLB. We observe an average 6.8% performance degradation due to static partitioning. Workloads combining applications with mixed MPKI values, particularly those including at least one high/medium-MPKI application (such as in W6, W8, and W10) suffer from a more severe performance drop. This is because static partitioning restricts the number of TLB entries available to high/medium-MPKI applications, thus reducing the ability of applications to accommodate increasing demands by taking up entries from others. STAR is also adaptable to scenarios with static partitioning, enabling two base addresses within the same instance or process to share a single TLB entry. Figure 20 also shows the performance of our approach on top of static partitioning. The results are normalized to the baseline shared TLB. STAR+static partitioning achieves an average of 14% performance improvement over static partitioning alone. This is because our approach is able to further optimize sub-entry utilization within individual processes, effectively increasing the TLB hit rate and enhancing overall performance.

F. Comparison with State-of-the-Art

The previous work MASK [9] addressed shared TLB contention in multi-application environments using TLB-Fill Tokens to manage how many warps can fill the shared TLB, and adjusted the TLB entries allocated to each application based on its L2 TLB miss rate, thus reducing thrashing. It also features a TLB bypass cache for entries from warps with insufficient tokens. Note that MASK includes optimizations

for both L2 cache and main memory interference. However, in our MIG baseline configuration, the L2 cache and memory are partitioned across different instances, which eliminates any interference between these two components. Therefore, our comparison focuses only on the TLB optimizations implemented in MASK. Figure 21 reports performance for STAR normalized to MASK, where our approach achieves an average 25% performance improvement over MASK. Although MASK effectively manages TLB contention through dynamic partitioning, it does not address the significant underutilization of sub-entries, which critically impacts performance.

VII. Related Work

Substantial prior studies have focused on address translation optimizations to improve system performance [5], [10], [11], [27], [31], [33], [41], [42], [55]. Several previous studies [10], [43] enhanced TLB hit rates by employing speculative techniques to predict the translations that miss in the TLBs. Many studies have delved into methods designed for enhancing page management to optimize the address translation process [2]–[4], [30], [32], [35], [57], [60]. An alternate set of techniques [12], [21], [24] improved TLB reach by generating contiguous translations. Research proposals also suggested an alternative memory management unit (MMU) cache structures, to cache multiple levels of the page tables [11], [14]. Additionally, the research community has explored approaches to increase TLB performance by using large pages and improving super-page management [19], [39]. Bharadwaj et al. [13] co-designed distributed TLBs with a lightweight interconnect to realize scalable shared L2 TLBs. Li et al. [33] optimized address translation in multi-GPUs through sharing and spilling aware TLB design. Compared to all the prior efforts, our research pioneers the optimization of MIG-enabled GPUs by innovatively addressing TLB thrashing and implementing a sharing mechanism for advanced TLB sub-entry designs.

VIII. Conclusion

In this paper targeting multi-instance GPUs, we comprehensively study the address translation efficiency in multi-tenant execution. Our investigation reveals that shared L3 TLB contention significantly impacts performance by increasing TLB thrashing and reducing the utilization of TLB sub-entries. To address this problem, we propose STAR that enables dynamic sharing of TLB entries among different base addresses. Experimental results demonstrate that STAR substantially enhances performance, delivering an average improvement of 28.7% across a variety of multi-tenant workloads.

Acknowledgment

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, #1910413, #2334628, and #2312157; and UGent-BOF-GOA grant No. 01G01421.

References

[1] NVIDIA Corp. (2022) NVIDIA H100 Tensor Core GPU Architecture. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core

[2] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, Feb 2015, pp. 354–365.

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

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

[5] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 457–468.

[6] H. AlJahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau, and J. Xu, "Multi-tenancy in cloud computing," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, 2014, pp. 344–351.

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

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

[9] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "MASK: Redesigning the GPU memory hierarchy to support multi-application concurrency," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 503–518.

[10] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *2011 38th Annual International Symposium on Computer Architecture*, 2011, pp. 307–317.

[11] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 48–59.

[12] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 237–248.

[13] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee, "Scalable distributed last-level TLBs using low-latency interconnects," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 271–284.

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

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54.

[16] Y. Dai, Y. Zhang, and X. Tang, "CEGMA: Coordinated elastic graph matching acceleration for graph matching networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture*, 2023, pp. 584–597.

[17] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *GPGPU-3: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, p. 63–74.

[18] S. Dong and D. Kaeli, "DNNMark: A deep neural network benchmark suite for GPUs," in *Proceedings of the General Purpose GPUs*, 2017, pp. 63–72.

[19] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, 2015, pp. 223–234.

[20] S. Feng, S. Pal, Y. Yang, and R. G. Dreslinski, "Parallelism analysis of prominent desktop applications: An 18-year perspective," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software*, 2019, pp. 202–211.

[21] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 178–189.

[22] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bimodal dram cache: Improving hit rate, hit latency and bandwidth," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 38–50.

[23] C.-C. Huang and V. Nagarajan, "Atcache: Reducing dram cache latency via a small sram tag cache," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 51–60.

[24] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015, pp. 66–78.

[25] Y. Kim, Y. Choi, and M. Rhu, "Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 607–612.

[26] I. King. (2017) Chipmakers Nvidia, AMD Ride Cryptocurrency Wave—for Now. [Online]. Available: www.bloomberg.com/news/articles/2017-07-17/chipmakers-nvidia-amd-ride-cryptocurrency-wave-for-now.

[27] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "Latr: Lazy translation coherence," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 651–664.

[28] J. Lee, J. M. Lee, Y. Oh, W. J. Song, and W. W. Ro, "SnakeByte: A TLB design with adaptive and recursive page merging in GPUs," in *2023 IEEE International Symposium on High-Performance Computer Architecture*, 2023, pp. 1195–1207.

[29] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "Miso: exploiting multi-instance GPU capability on multi-tenant GPU clusters," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 173–189.

[30] B. Li, Y. Guo, Y. Wang, A. Jaleel, J. Yang, and X. Tang, "IDYLL: Enhancing page translation in multi-gpus via light weight PTE invalidations," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1163–1177.

[31] B. Li, Y. Wang, and X. Tang, "Orchestrated scheduling and partitioning for improved address translation in gpus," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[32] B. Li, J. Yin, A. Holey, Y. Zhang, J. Yang, and X. Tang, "Trans-FW: Short Circuiting Page Table Walk in Multi-GPU Systems via Remote Forwarding," in *Proceedings of the 29th International Symposium on High-Performance Computer Architecture*, 2023.

[33] B. Li, J. Yin, Y. Zhang, and X. Tang, "Improving address translation in multi-GPUs via sharing and spilling aware TLB design," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1154–1168.

[34] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM Transactions on Design Automation of Electronic Systems*, vol. 19, no. 3, pp. 1–24, 2014.

[35] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccNUMA systems," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 90–99.

[36] NVIDIA, "NVIDIA Driver Documentation - NVIDIA Multi-Instance GPU User Guide," 2023. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/

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

[38] I. Odun-Ayo, S. Misra, O. Abayomi-Alli, and O. Ajayi, "Cloud multitenancy: Issues and developments," in *Companion Proceedings of The10th International Conference on Utility and Cloud Computing*, 2017, p. 209–214.

[39] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 679–692.

[40] M. Parasar, A. Bhattacharjee, and T. Krishna, "SEESAW: Using superpages to improve VIPT caches," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 193–206.

[41] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017, pp. 444–456.

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

[43] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015, pp. 1–12.

[44] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, p. 258–269.

[45] G.-G. S. Pouchet L-N. (2010) Polybench: The polyhedral benchmark suite. [Online]. Available: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[46] B. Pratheek, N. Jawalkar, and A. Basu, "Improving GPU multi-tenancy with page walk stealing," in *2021 IEEE International Symposium on High-Performance Computer Architecture*, 2021, pp. 626–639.

[47] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular GPU applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018, pp. 180–192.

[48] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing," in *2016 IEEE International Symposium on Workload Characterization*, 2016, pp. 1–10.

[49] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "MGPUSim: Enabling multi-GPU performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, p. 197–209.

[50] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," *ACM SIGPLAN Notices*, vol. 29, no. 11, pp. 171–182, 1994.

[51] X. Tang, Z. Zhang, W. Xu, M. T. Kandemir, R. Melhem, and J. Yang, "Enhancing address translations in throughput processors via compression," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, p. 191–204.

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

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

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

[55] A. Bhattacharjee, "Translation-triggered prefetching," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 63–76.

[56] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware TLBs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 698–710.

[57] Y. Wang, B. Li, A. Jaleel, J. Yang, and X. Tang, "GRIT: Enhancing multi-GPU performance with fine-grained dynamic page placement," in *2024 IEEE International Symposium on High-Performance Computer Architecture*, 2024, pp. 1080–1094.

[58] K. Wood and M. Anderson, "Understanding the complexity surrounding multitenancy in cloud computing," in *2011 IEEE 8th International Conference on e-Business Engineering*, 2011, pp. 119–124.

[59] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, p. 174–183.

[60] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 331–345.

[61] J. Yi and Y. Lee, "Heimdall: Mobile GPU coordination platform for augmented reality applications," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.

[62] F. Yu, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu, and X. Chen, "Automated runtime-aware scheduling for multi-tenant DNN inference on GPU," in *2021 IEEE/ACM International Conference On Computer Aided Design*, 2021, pp. 1–9.

[63] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, "Tunnels for bootlegging: Fully reverse-engineering GPU TLBs for challenging isolation guarantees of NVIDIA MIG," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 960–974.

[64] S. Zhao, H. Zhang, C. S. Mishra, S. Bhuyan, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. Das, "Holoar: On-the-fly optimization of 3d holographic processing for augmented reality," in *Prodeedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 494–506.