Barre Chord: Efficient Virtual Memory Translation for Multi-Chip-Module GPUs

Yuan Feng*, Seonjin Na†, Hyesoon Kim†, Hyeran Jeon*
*University of California, Merced †Georgia Institute of Technology
{yfeng44, hjeon7}@ucmerced.edu, seonjin.na@gatech.edu, hyesoon@cc.gatech.edu

***Abstract*—With the advancement of processor packaging technology and the looming end of Moore's law, multi-chip-module (MCM) GPUs become a promising architecture to continue the performance scaling. However, due to the increasing concurrency, it is challenging to achieve scalable performance. In this study, we show that the limited parallelism in IOMMU is one of the critical bottlenecks and propose *Barre Chord* to fundamentally reduce the translation loads. By leveraging the unique GPU execution model and page mapping on MCM-GPUs, *Barre* translates virtual addresses in a unit of *coalescing group*. Once one page is translated, all the other pages within the same coalescing group can be translated with simple *calculations* without page table walks. *Full Barre (F-Barre)* further reduces translations by enabling *intra-MCM translation* through coalescing information sharing across GPU chiplets and *contiguity-aware coalescing group expansion*. With the combination of Barre and F-Barre, the Barre Chord outperforms state-of-the-art solutions by an average of 1.36× (2.09× with coalescing group expansion) with negligible area overhead (4.22% of a GPU L2 TLB).**

I. Introduction

With the packaging technology evolution, Multi-Chip-Module (MCM) GPUs have been introduced to continue the performance scaling in the post-Moore era. Though MCM design can help increase computing power, there are new challenges such as the non-uniform memory access (NUMA) effect and the increased burden of virtual memory translation. Several studies have effectively reduced NUMA effect through various page mapping strategies and remote data caching and coalescing [3], [8], [20], [40]. Though these solutions indirectly reduce the address translation overhead by enabling GPU chiplets to find more data and translations from local memory, we still observe enough room to improve performance. For example, Fig 1 shows an almost linear speedup with more page table walkers (PTWs). However, when we use *infinite* PTWs, the speedup is saturated to around 2×. This is because adding more PTWs is only effective in reducing PTW queueing delay while leaving the other latencies (e.g., page table walk and IOMMU access) in the translation process unchanged. There should be a more comprehensive solution that considers the entire translation process and fundamentally reduces translation costs.

In this paper, we propose *Barre Chord*[1] that introduces a novel translation method for MCM-GPUs. Barre Chord consists of *Barre* and *Full-Barre*[2], each *removes* page table walks and IOMMU accesses from the translation process, respectively. Barre rethinks the virtual memory translation in the context of MCM-GPU and reduces the address translation overheads by exploiting the unique GPU execution model and data mapping patterns. An MCM-GPU consists of multiple GPU chiplets; each chiplet has its local memory. In an MCM-GPU, applications process a large data on multiple GPU chiplets. The large data allocates many pages across GPU chiplets and each page needs to be mapped from virtual space to physical space individually. Barre takes the unique opportunity of such multi-GPU-chiplet processing to map the pages much more structurally, which enables skipping many page table walks fundamentally.

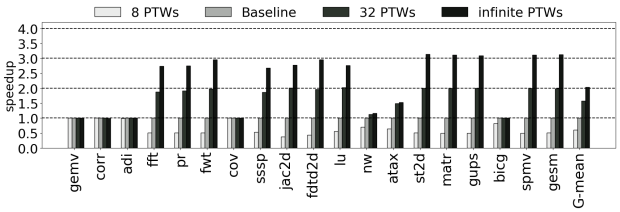


Fig. 1: Speedups with 8, 16, 32, and infinite PTWs

For an application, GPU chiplets run the same program code. Thus, the pages of each data tend to be accessed in similar timings across GPU chiplets. By leveraging this fact, Barre proposes to map the pages of each data on the same physical locations across GPU chiplets so that their addresses can be translated together. We call those pages mapped on the same physical address across GPU chiplets as a *coalescing group*. In other words, the pages within the same coalescing group have the same physical address except for the GPU chiplet ID. Therefore, once we know a page's physical address, we can *calculate* the physical address of all the other pages in the same coalescing group, without expensive page table walks. The coalescing group is created at the time of page allocation and is mapped to any physical pages that are commonly available across GPU chiplets. If there are not commonly available pages, the pages are mapped individually as in the conventional page mapping scheme. The coalescing group information is encoded in the unused bits in the page table entry (PTE).

Super pages [5], [24], [37] also allocate large data into physical space systemically. However, a super page is required to map on *consecutive* large physical space. Barre

[1]A guitar chord that presses down multiple strings over a fret with a finger.
[2]A special type of Barre Chord that presses all six strings with a finger.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00065

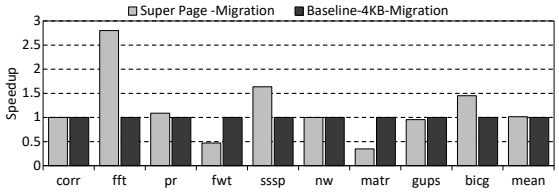


Fig. 2: Speedup with 2MB super page when migration is enabled

does not have such restrictions and it coalesces pages *across* GPU chiplets only for passing address translation information among multiple pages. Also, the performance benefit of super pages is limited in an MCM-GPU due to distributed memory mapping. With super page, a few hot pages will be mapped on fewer GPU chiplets and lead to more remote memory accesses and migrations [41]. As shown in Fig 2, some applications (`fwt` and `matr`) experience a significant performance drop due to increased remote memory accesses and higher page migration penalties.

Besides super page, address translation overhead has been tackled with various other solutions such as L2 TLB sharing [8], [27] and translation prefetch on single- and multi-GPU platforms. The impact of L2 TLB sharing is diminishing in MCM-GPUs because advanced page mapping algorithms [20], [30], [41] reduce remote data accesses; thus, less chance of finding translations in remote TLBs. The increased concurrency in an MCM-GPU drops prefetching accuracy because the translation requests lose patterns as shown in Fig 5. In the CPU domain, hashed page table has been used to enable calculation-based page translation. But, the hashed page tables have issues of hash collision, impracticality of sharing pages by multiple processes, and inflexible page mapping. Barre does not have such problems.

While Barre removes many page table walks, each translation mandates an IOMMU access. To reduce the hassle of IOMMU access, Barre Chord proposes *Full Barre (F-Barre)*, which enables intra-MCM translation. F-Barre extends Barre by allowing GPU chiplets to share the *coalescing group information*. With the coalescing group information, addresses can be calculated without accessing IOMMU. Inter-GPU TLB sharing [8], [27] can also reduce IOMMU accesses. However, it mandates remote GPU chiplet access upon every TLB misses. On the other hand, F-Barre does not need to access remote GPU chiplet if the TLB has at least one page's address that is in the same coalescing group with the requested page. Note that when GMMU is used instead of IOMMU, F-Barre can also remove GMMU accesses through this coalescing information sharing. F-Barre also opportunistically expands coalescing groups. When multiple coalescing groups are mapped on consecutive physical pages, F-Barre merges them such that one translation can calculate all pages in the enlarged coalescing group.

Barre outperforms two state-of-the-art solutions [8], [27] by an average of 12.8% up to 41%. F-Barre escalates the average speedup to 1.36× (2.09× with coalescing group expansion).

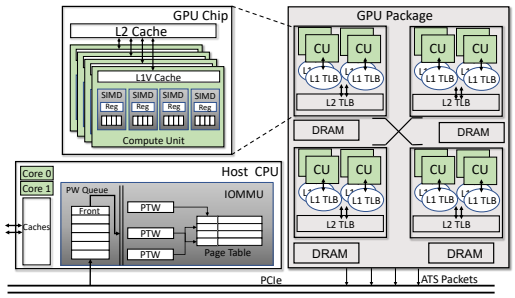


Fig. 3: Baseline MCM-GPU architecture

Our contributions are as below.

- We show that limited parallelism in IOMMU is one of the critical performance bottlenecks in MCM-GPUs and analyze the challenges in applying conventional solutions to MCM-GPUs.
- We propose a novel virtual memory translation method, *Barre Chord*, which fundamentally reduces the demands for page table walks by exploiting the unique architecture and page mappings of MCM-GPUs.
- The proposed Barre Chord introduces a novel coalescing group-level calculation-based translation, which provides better performance while not sacrificing the flexibility of page table-based translations.

II. Background

A. MCM-GPU architecture and address translation

Fig 3 shows a typical architecture of modern MCM-GPUs. A CPU is connected to an MCM-GPU through PCIe. The MCM-GPU has multiple GPU chiplets that are connected via a high-speed interconnect network. Each GPU chiplet has a dedicated memory. CPU and MCM-GPU use a flat virtual address. The unified virtual memory is implemented with an IOMMU residing in the host CPU to govern the translations of all IO devices including MCM-GPU [8], [27], [40]. Computing units (CUs) have a private L1 TLB and a shared L2 TLB. Upon L2 TLB miss, an *address translation service* (ATS) packet is sent to the IOMMU through PCIe. Page table walks are handled by 8-16 threads [8], [27] with multiple *page table walkers* (PTWs). The requests are enqueued to a *page walk queue* (PW-queue) to be served by PTWs.

B. Page mapping policies in MCM-GPU

In an MCM-GPU, as each GPU chiplet has a local memory and can access remote GPU chiplet's memory, memory access latency is non-uniform. Various page mapping algorithms have been proposed to reduce the lengthy remote memory accesses among GPU chiplets [3], [7], [20], [30], [40]. LASP [20] considers locality when mapping pages to minimize the remote memory accesses. LASP analyzes data access pattern at compile time and maps consecutive a few pages in either row- and column dimension on the same GPU chiplet. The cooperative thread arrays (CTAs) that access those pages are co-located on the same GPU chiplet to reduce remote memory accesses. This locality information is used by the GPU driver

TABLE I: Benchmarks

Benchmark suite	Abbr.	App Name	L2TLB MPKI	Category
polybench	gemv	gemver	0.015	low
polybench	corr	correlation	0.045	low
polybench	adi	adi	0.051	low
Shoc	fft	fft	0.48	low
HeteroMark	pr	pagerank	0.828	low
AMD APP SDK	fwt	fastwalshtransform	2.27	mid
polybench	cov	covariance	3.24	mid
Panotia	sssp	sssp	3.38	mid
polybench	jac2d	jacobi2d	4.78	mid
polybench	fdtd2d	fdtd2d	10.12	mid
polybench	lu	lu	17.14	mid
Rodinia	nw	nw	21.56	mid
polybench	atax	atax	34.28	mid
Shoc	st2d	stencil2d	46.90	mid
AMD APP SDK	matr	matrixtranspose	174.99	high
MAFIA	gups	gups	724.80	high
polybench	bicg	bicg	2128.63	high
Shoc	spmv	spmv	3835.95	high
polybench	gesm	gesummv	4762.86	high

when mapping virtual pages to GPU chiplets. LASP enforces CTA and page mapping across GPU chiplets. Within each GPU chiplet, the assigned CTAs are mapped across CUs as the execution progresses. The kernel-wide chunking [30] maps consecutive pages and CTAs on the same GPU chiplet by assuming a coarse-grained locality. Unlike LASP, it does not use compiler support and can work as a runtime system with unmodified, single-GPU optimized applications, while admittedly giving up some locality opportunities. CODA [21] is another compiler-assisted solution and co-locates data and CTA. For the linearly accessed data, consecutive pages and CTAs are mapped on the same GPU chiplet, similar to LASP. But, unlike LASP, CODA maps sparse or irregularly accessed data in a round-robin fashion across GPU chiplets. Another well-studied on-demand paging [3], [11] captures the runtime locality. While flexible, runtime page migrations incur expensive page faults. While Barre is compatible with the on-demand paging (Section VI), we focus on the virtual memory translation efficiency by assuming that page mapping is determined earlier to avoid page fault overhead, similar to previous works [8], [20], [27], [39], [41]–[43], [47], [51].

C. Contiguity-based page and TLB coalescing

Super pages have been used to mitigate the virtual memory translation overhead as each translation can cover more addresses. However, its strict data mapping enforcement raises issues such as a limited number of available large pages, memory fragmentation, and high page fault overhead [5], [7], [22], [26]. As a flexible solution, TLB coalescing has been proposed [2], [5], [24], [34], [36], [37]. TLB coalescing dynamically coalesces multiple virtual-to-physical page mappings into single TLB entries. However, TLB coalescing still relies on the temporal but contiguous physical frame number (PFN) allocations, which has limited opportunities in MCM-GPUs where each GPU chiplet has a distinct local memory.

III. Challenges of Efficient Virtual Memory Translation in MCM

A. Methodology

We design an AMD GCN3-like MCM-GPU that has four GPU chiplets in a cycle-level simulator, MGPUSim [44], as illustrated in Fig 3. We have carefully configured various

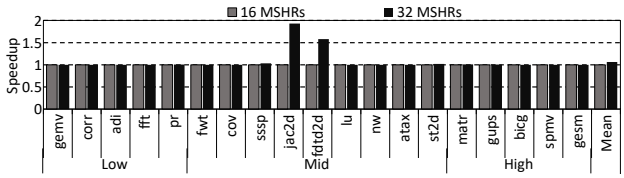


Fig. 4: Performance impact of L2TLB MSHRs.

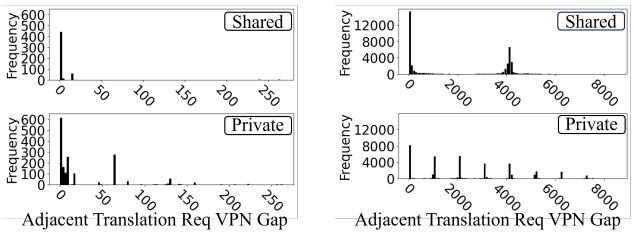
parameters according to the earlier studies (full configurations in Table II). The IOMMU uses 16 PTWs and a 48-entry PW-queue according to several studies [8], [26], [27], [43]. Our baseline does not use GMMU by following the configurations of earlier studies [7], [8], [18], [27], [44]. We discuss how Barre Chord can be integrated with GMMU (Sec VI) and evaluate it over the state-of-the-art GMMU solution (Sec VII-F). We use physically indexed and physically tagged caches [3], [8], [24]. LASP [20] is used as a baseline page mapping algorithm. We use 19 applications having diverse IOMMU intensity (measured by *L2 TLB Misses Per Kilo warp-level Instructions*) as specified in Table I. The applications are selected from PolyBench [32], Shoc [12], HeteroMark [45], AMD SDK [15], Panotia [9], Rodinia [10], and MAFIA [19].

B. Address Translation Efficiency in MCM-GPU

To understand the performance impact of address translation, we measure speedup while increasing the number of PTWs. Fig 1 shows that the majority of applications have almost linear speedups with more PTWs, except for some applications with low IOMMU intensity. This means that the limited parallelism in IOMMU is one of the critical performance bottlenecks. To isolate the impact of IOMMU from that of intra-MCM resources, Fig 4 shows normalized performance with more L2 TLB MSHRs. On average, doubling the MSHRs only brings 6% performance benefits, where the vast majority of applications do not see any speedup. This reveals that the bottleneck is not the capability to hold the outstanding translation misses but the capability to process them. Increasing parallelism (e.g., adding more PTWs) will incur area and power overhead, and also it is not a scalable solution. The following subsections discuss the challenges in applying conventional solutions to MCM-GPU.

C. Can PTE prefetch or super page help?

To solve the address translation overhead, several studies have proposed translation prefetch and large and flexible-sized pages for single GPU or CPU systems [5], [24], [37]. Those solutions require two types of contiguity 1) among requested virtual page numbers (VPNs) and 2) in VPN to PFN mapping. According to our experiments, such contiguity is rarely observed in MCM-GPUs due to increased concurrency and distributed memory. To understand the translation contiguity in an MCM-GPU, we design a *hypothetical* L2 TLB that is shared across GPU chiplets and compare it with our baseline MCM-GPU having private TLBs. Fig 5 shows the VPN gap between any two consecutive translation requests received by IOMMU. With private L2 TLBs, highly scattered addresses



(a) Corr (b) PageRank

Fig. 5: VPN gap distribution

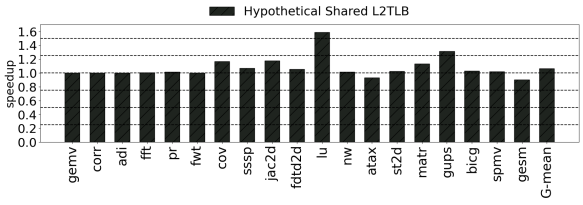


Fig. 6: Speedup of shared L2 TLB over private L2 TLBs

are requested as reflected by more and irregularly presenting spikes. Therefore, it is more challenging to predict the next address for translation. Likewise, as pages are mapped across distinct GPU memories, the mapping between VPN and PFN is less contiguous. Thus, it is not easy to apply super pages. Also, as super pages mandate a coarser-grained page mapping, a few hot pages will intensify the remote memory accesses (Fig 2) and will offset the advantage of page allocation algorithms. Therefore, a new approach is necessary.

D. Can L2 TLB sharing help?

Several recent studies have optimized address translation with inter-GPU L2 TLB sharing [8], [27]. To understand the impact of TLB sharing under advanced page allocation, we measure the performance of an oracle case inter-L2 TLB sharing by using a hypothetical shared L2 TLB with LASP page mapping. The shared L2 TLB is configured with 4× more entries while not increasing access latency, and 4× more processing bandwidths than private TLBs. This *ideal* shared L2 TLB excludes the inter-GPU communication overheads as it is one physical TLB and reduces the impacts of conflict/capacity misses with large and high-bandwidth TLB. As shown in Fig 6, the oracle case shows an average of 6% speedup over our baseline. This result means that there are still some shared translations among GPU chiplets due to common input data. However, less than half of the applications show speedup. As this result is the ideal speedup of the TLB sharing with an advanced page allocation, the actual speedup will be less. Therefore, we need a different solution than TLB sharing.

IV. Barre

Barre optimizes the address translation process in MCM-GPU by opportunistically skipping page table walks. For a GPU application, all CTAs run the same program code while processing different parts of each input data. Thus, the parts of each input data are mapped at a similar timing across GPU chiplets. Barre *coalesces* these address translations for the same input data and allows skipping expensive page table walks once one page in each *coalescing group* (Section IV-A) is translated.

In an MCM-GPU, CTAs and their input data are mapped across GPU chiplets [3], [20]. Due to the distributed mapping, it is challenging to coalesce the translations. We solve this problem by enforcing the same data to be mapped on the same local physical frame numbers (PFNs) in each GPU's memory such that the PFN can be *calculated* once the coalescing information is retrieved.

A. Coalescing Group

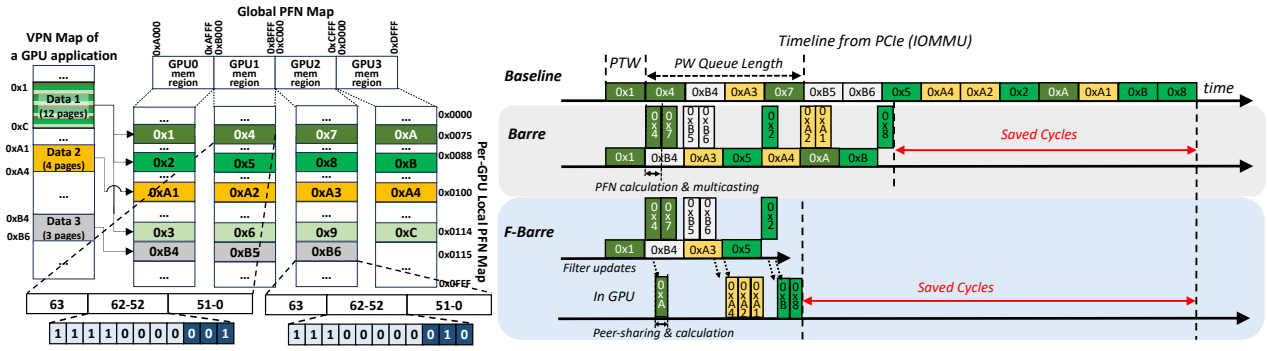
A coalescing group is a batch of pages of one data (allocated by one GPU malloc API execution e.g., one matrix) that are mapped on the same local PFNs across multiple GPU chiplets. A coalescing group consists of 2 to N pages, where N is the number of GPU chiplets in the MCM-GPU. If the page allocation algorithm (LASP [20] in our case) maps multiple pages for a data on each GPU, we partition a data into multiple coalescing groups. The same colored pages in Fig 7a are in the same coalescing group. The 12 virtual pages of data 1 are mapped over three coalescing groups.

B. Barre Operational Overview

Suppose we allocate three data having 12, 4, and 3 pages each, as illustrated in Fig 7a. Based on LASP, a consecutive few virtual pages are mapped to each GPU chiplet to gain a higher locality. Each GPU chiplet has three pages for data 1 (VPNs 0x1 to 0x3 on GPU_0, 0x4 and 0x6 on GPU_1, etc.) and one page for data 2 (VPNs from 0xA1 to 0xA4 on GPU_0 through GPU_3 each). Without Barre, each page needs one translation separately; a total of 19 translations for the three data. With Barre, the pages in the same coalescing group can be served by one translation. The same colored pages in Fig 7a are in the same coalescing group. Data 1 consists of three coalescing groups. Data 2 and 3 have one coalescing group each. Thus, a total of five translations can cover the 19 pages.

Once a page is translated, Barre can *calculate* all the other PFNs in the same coalescing group. Thus, the expensive PTWs can be skipped. Barre can speculatively calculate and send all the other PFNs of the coalescing group to corresponding GPUs upon one translation. However, our experiments show this multicasting drops performance due to the limited outbound bandwidth of IOMMU. Thus, we configure Barre to cover the translations for the pending requests only. If all pages in the same coalescing group are requested at similar times, one translation can cover all pages.

Suppose that the ATS requests are received and handled by IOMMU as shown in Fig 7b. In Barre, one of the same colored pages in the pending queue takes the full latency of IOMMU operation (e.g., 0x1 in green) while the PFNs of the following pages of the same color (e.g., 0x4 and 0x7) are calculated behind the scenes. Barre cuts the total translation latency by over half. While handling a translation, IOMMU checks the PW-queue to find pending requests in the same coalescing



(a) VPN-to-PFN mapping for 3 data

(b) Walkthrough: Translation speedup when PW-Queue has 4 entries.

Fig. 7: Barre Chord (a) Page Mapping and (b) Potential Speedup.

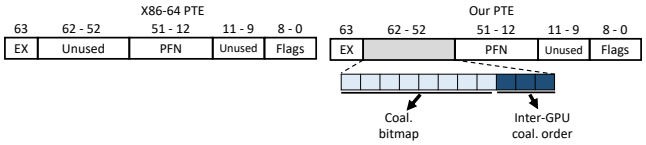


Fig. 8: Proposed PTE format

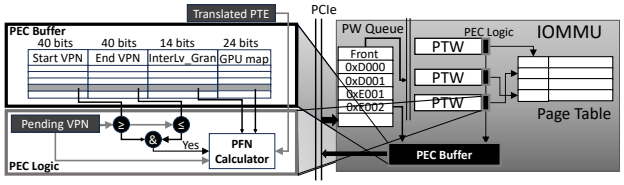


Fig. 9: PTE logic and PFN calculator

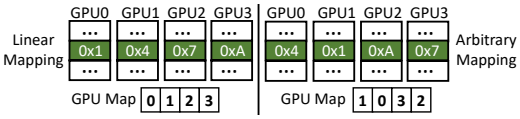


Fig. 10: GPU map example for Data 1 in Fig 7a

group and calculates the PFN by using the coalescing group information retrieved from the translated PTE.

C. Data Mapping Enforcement

To cover multiple pages with one translation, Barre maps the pages in the same coalescing group to the same local PFNs across GPUs. If multiple pages are mapped per GPU, the pages in the same order are mapped to the same local PFN e.g., 0th VPNs in all GPUs are mapped to 0x75. This enforcement will be applied to the data that are allocated with a GPU malloc API. The available local PFNs can be located by the GPU driver (e.g., `amdgpu_hmm_range_get_pages(.)` in AMD GPU driver). **Example 1:** In Fig 7a, according to LASP, data 1 needs to map three consecutive VPNs per GPU. Suppose that the driver finds three commonly available PFNs across GPUs; 0x75, 0x88, and 0x114. Then, the first three virtual pages, 0x1, 0x2, and 0x3 are mapped on GPU_0's local PFNs 0x75, 0x88, and 0x114, respectively. Likewise, the second three pages, 0x4, 0x5, and 0x6 are mapped on GPU_1's local PFNs 0x75, 0x88, and 0x114, respectively. If the starting global PFNs of each GPU's memory are 0xA000, 0xB000, 0xC000, and 0xD000, data 1 is mapped on 0xA075, 0xA088, 0xA114, 0xB075, 0xB088, 0xB114, etc. This mapping enforcement also supports multi-application (Section VII-I).

D. Page Table Entry Revision

The coalescing group information is encoded in the unused bits of x86 64-bit PTE format, as illustrated in Fig 8. The *coal_bitmap* uses binary values where 1 indicates GPU participation and 0 indicates non-participation. The current design supports up to eight GPUs with eight bits. In the *inter-GPU_coal_order*, numeric values like 0, 1, and 2 indicate the position of the page within the coalescing group, such as 0th VPN, 1st VPN, or 2nd VPN. **Example 2:** For the gray coalescing group in Fig 7a, the coal_bitmap has binary value `11100000` because first three GPUs are involved in this coalescing. The inter-GPU_coal_order value is different for each PTE. For 0xB6, inter-GPU_coal_order is `010` (= numeric value 2) as it is the 2nd VPN (from 0th) in the coalescing group.

E. PEC Logic

To decode the coalescing information from a PTE and calculate the PFNs of pending requests, we design *Page Entry Coalescing (PEC) logics* in the IOMMU, as illustrated in Fig 9. A PEC logic has comparators and a *PFN calculator* and uses a shared *PEC buffer* that maintains the starting and ending VPNs, the page interleaving granularity (*interlv_gran*; the number of consecutive VPNs mapped per GPU chiplet according to LASP), and the VPN-to-GPU-chiplet mapping (*GPU_map*) of each data. The PEC buffer has five entries as all of our benchmark applications use up to five large data. When the table is full, a new data overwrites an entry having smaller data's information. Each PTW has a dedicated PEC logic.

Though Fig 7a shows an intuitive example where VPNs are linearly mapped across GPU chiplets, the mapping can be an arbitrary order. The GPU_map indicates the GPU chiplet ids that the VPNs are mapped to; 0, 3, and 2 indicate that the 0th to 2nd VPNs are mapped on GPU_0, GPU_3, and GPU_2, respectively. Fig 10 shows the examples when pages

are mapped linearly (left) and arbitrarily (right) on GPUs. 0x1 is the 0th VPN in the coalescing group and is mapped on GPU_0 in the left example so the first entry in GPU_map has value 0, while it is mapped on GPU_1 in the right example so the entry in GPU_map has value 1. Note that this mapping order is determined when CTAs are scheduled. According to LASP, a consecutive few CTAs (either in row or column order) and their pages are mapped together on the same GPU chiplet to enforce locality. Thus, VPNs are consecutively mapped either within each GPU or across GPUs. This enables all the coalescing groups of a data to see the same VPN-to-GPU mapping order (either ascending or descending) regardless of the value gap between any two neighboring VPNs. Thus, we allocate one GPU_map per data. With this GPU_map, we can locate the GPU chiplet and the global PFN that each VPN is mapped to.

Example 3: The PEC buffer entry for data 1 in Fig 7a has Start and End VPNs as `0x1` and `0xC` because data 1's VPNs are from 0x1 to 0xC. The interlv_gran is `3` because each GPU has three pages for data 1. GPU_map is `000001010011...` (= every three bits indicate each VPN's GPU id; numeric values are 0, 1, 2, 3,...) because the coalescing groups of data 1 map 0th VPN to GPU_0, 1st VPN to GPU_1, 2nd VPN to GPU_2, and 3rd VPN to GPU_3. Only the first 12 bits are considered as only four GPUs are involved in this coalescing, which can be checked with coal_bitmap in PTE.

F. PFN Calculation

Once an ATS is enqueued in the PW-queue, an available PTW dequeues it and translates the requested virtual address through a page table walk. Once a PTW translates a PTE, its PEC logic checks if the PTE has more than one 1's in the coal_bitmap. If so, the PEC logic retrieves the data information from the PEC buffer by using the PTE's VPN. Then, PEC looks up the PW-queue to find the pending requests belonging to the same data (pending VPN is within the start and end VPN range of the retrieved coalesced data). Once found, the pending request and the translated PTE are passed to the PFN calculator. The pending VPN is checked if it is within the same coalescing group by calculating candidate *coalescing VPNs*. Because each GPU maps interlv_gran pages per data and we enforce the pages of the same data to be mapped on the same local PFNs across GPU chiplets, any two pages within the same coalescing group have a VPN gap as a multiple of interlv_gran. Thus, PEC logic can calculate the coalescing VPNs by decrementing or incrementing PTE's VPN by interlv_gran. The number of coalescing VPNs will be the bit count in coal_bitmap. If the pending VPN matches with any coalescing VPNs, the request is in the same coalescing group. Now, we can decipher the local PFN of this pending VPN because all pages in the coalescing group have the same local PFN.

To figure out the global PFN, we check the relative position (inter-GPU_coal_order) of this page. The relative position of the pending page can be estimated by checking its position in the coalescing VPNs that are calculated

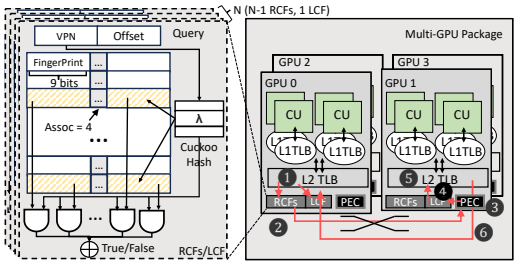


Fig. 11: Intra-MCM translation with Cuckoo filters

above. If the pending VPN is 0th VPN in the coalescing VPNs, the inter-GPU_coal_order is 0. By checking the $(\text{inter-GPU\_coal\_order}_{pending})_{th}$ value in the GPU_map, we can identify the GPU chiplet and the global starting PFN that this page is mapped to. The base PFN of each GPU chiplet memory (global PFN map in Fig 7a) is available in IOMMU.

Example 4: In Fig 7a, suppose a PTW finishes translating VPN 0x4 and finds that the PFN is 0xB075. The PTW looks up the PW-queue and finds 0xA. As 0xA is within data 1's VPN range (from 0x1 to 0xC), PEC logic further checks if 0xA is in the same coalescing group. As the interlv_gran of data 1 is `3`, 0x4 is incremented by 3 up to two times or decremented up to once because 0x4's inter-GPU_coal_order is 1, which means there are two greater and one smaller VPNs in the coalescing group. By adding 3 twice, the result becomes matching with 0xA. Thus, 0xA is in the same coalescing group. So, we now know that 0xA's local PFN is 0x75. As 0x4 is incremented by two times where 0x4 is the 1st VPN in the group, 0xA is 3rd VPN in the group. Thus, 0xA checks the 3rd value in GPU_map, which is 3 (=`011`) (see Example 3). As the GPU_3's base PFN is 0xD000, the 0xA's PFN is 0xD075, as can be found in Fig 7a.

G. Driver Modification

We enforce the data mapping by modifying LASP in the GPU driver. Within the LASP-enabled malloc function, we replace the standard memory allocation with our algorithm. We iterate the available PFNs of one GPU chiplet and check if the PFN is also available in the sharer chiplets. The algorithm stops when it finds interlv_gran PFNs per GPU. Then, we create a coalescing group per PFN. The coalescing group information is encoded in the PTE. Once the pages are mapped on the GPUs, the PEC buffer information is recorded. If the algorithm cannot find commonly available pages across the sharer GPUs, we fall back to the driver's default memory allocation.

V. Full Barre

Barre can effectively reduce page table walks while having three limitations: 1) every translation needs an IOMMU access, 2) the coalescing group size is limited to the number of sharer GPUs, and 3) the address oblivious PTW scheduling drops the opportunities of coalesced PFN calculation. To fundamentally reduce ATS traffic and maximize the opportunity for coalesced PFN calculation, we propose *Full Barre (abbrev. F-Barre)*.

	GPU 0			GPU1		
Step	L2TLB	LCF	Action	L2TLB	RCF0	Action
0	0xA1		ATS return			
1	0xA1	0xA1	update LCF			
2	0xA1	0xA1			0xA1, 0xA2	update RCF
3	0xA1	0xA1		Miss 0xA2	0xA1, 0xA2	RCF hit 0xA2
4	0xA1	0xA1	PEC(0xA2)		0xA1, 0xA2	
5	0xA1	0xA1	LCF hit 0xA1		0xA1, 0xA2	
6	0xA1	0xA1	TLB visit		0xA1, 0xA2	
7	0xA1	0xA1	0xA2 Comp.		0xA1, 0xA2	
8	0xA1	0xA1		0xA2	0xA1, 0xA2	Remote Hit

Fig. 12: A walkthrough of F-Barre when GPU_0 and GPU_1 are in a coalescing group for pages 0xA1 and 0xA2: **[steps 0-2]** GPU_0 receives a translation for 0xA1 and updates its LCF and GPU_1's RCF_0. **[steps 3-4]** GPU_1 looks up TLB, LCF, and RCF_0 for 0xA2 and RCF hits and TLB and LCF miss; sends a request to GPU_0. GPU_0 calculates coalescing VPNs from the requested 0xA2 with PEC logic. **[steps 5-7]** GPU_0 finds 0xA1 hits in LCF and looks up TLB with 0xA1. Then, it calculates the PFN of 0xA2 by using coalescing information of 0xA1. **[step 8]** GPU_1 receives and inserts the PFN to TLB.

A. Intra-MCM Translation

An MCM-GPU allows high-bandwidth communications among GPU chiplets within it. Thus, it is more efficient to share translations within the MCM-GPU package rather than accessing IOMMU via slow PCIe. We propose to share *coalescing information* among GPU chiplets to enable coalesced PFN calculations within the MCM-GPU package. This intra-MCM translation has three challenges: how to know 1) *which TLB entries* are within the same coalescing group?, 2) *which chiplet* has such TLB entry?, and 3) how to translate addresses within MCM-GPU? We address these below.

1) ***Sharer prediction****:* To locate the sharer GPU chiplet that has the information for translation, we propose *sharer prediction*. The TLB entries that have the VPNs in the same coalescing group (we call these *coalescing VPNs*) can provide the information for translation. Thus, the predictor should be able to indicate the existence of the coalescing VPNs in each GPU chiplet's TLB. This requires the predictor to be updatable upon TLB insertion and eviction. To fulfill this requirement, we use *cuckoo filter* [13]. The cuckoo filter is lightweight and supports both insertion and deletion of items. However, the cuckoo filter cannot *identify* a GPU chiplet because it indicates the residency of an item via binary output either `hit` or `miss`. Thus, we propose to employ as many filters as the number of sharer chiplets. We can locate the peer GPU chiplet that has the coalescing VPNs by checking all the filters and identifying the one that `hits`. We call these filters as *remote coalescing group filters (RCFs)*.

Unlike TLB sharing solutions, as we aim to find a *coalescing (not exact)* TLB entry, when L2 TLB misses, it is possible that the local TLB may have the coalescing VPN while it does not have the exact requested VPN. If the local TLB has the coalescing VPN, the requested VPN can be translated within the GPU chiplet. Thus, upon L2 TLB miss, each GPU chiplet also searches its TLB to find coalescing VPNs. As each chiplet also needs to look up its TLB when remote chiplets send requests, to avoid any contention between regular TLB accesses from coalescing VPN searches, we employ another filter namely *local coalescing group filter (LCF)*. According to our CACTI measurement, filters take only 1.7% of a TLB access power. Thus, for the coalescing VPN search, each GPU chiplet checks the LCF first and checks the TLB when LCF `hits`.

Example 5: In Fig 11, GPU_0 searches a VPN in its TLB, LCF, and RCFs in parallel (❶). If TLB and LCF miss and an RCF hits (e.g., RCF_1), the GPU chiplet sends a request to the peer chiplet, GPU_1 (❷). Then, GPU_1 calculates the coalescing VPNs (❸) and checks them with its LCF (❹). Once LCF hits, GPU_1 looks up its TLB (❺), calculates the PFN with the coalescing information, and sends the PFN to GPU_0 (❻).

2) ***Filter update with coalescing VPNs****:* Each GPU chiplet (say GPU_{id}) is responsible for updating its LCF and RCF_{id}s in all the other sharer GPUs whenever a TLB entry is added or evicted. As the goal is to search coalescing VPNs, F-Barre updates RCFs with the exact VPN as well as the coalescing VPNs. In Fig 12, once GPU_0 receives an ATS response for VPN 0xA1, it adds 0xA1 to its LCF and sends *filter update* messages to the peer chiplets to add 0xA1 *and 0xA2* to their RCF_0 because 0xA2 is in the same coalescing group. Then, when a peer chiplet needs a translation for 0xA2, the RCF_0 will hit. This enables peer chiplets to search by any VPN within the same coalescing group *without knowing the exact VPN in GPU_0's TLB*. When 0xA1 is evicted from L2 TLB, GPU_0 will delete 0xA1 from LCF and send the filter update messages to the peer GPU chiplets to remove 0xA1 and 0xA2 from their RCF_0.

GPU chiplets update RCFs with newly inserted TLB entry's VPN as well as coalescing VPNs. LCFs are updated with the newly inserted entry's VPN only to reflect the actual TLB contents. The number of coalescing VPNs is the number the bits set in coal_bitmap in the TLB entry. The filter update message consists of a 1-bit *command* (add or delete), a 3-bit *sender GPU chiplet id*, and a 40-bit *coalescing VPN* (a total of 43 bits). The filter update message delivery is on a best-effort basis (no acknowledgment required) and not in the critical path (Section VII-E has perf evaluation).

3) ***Coalescing VPN identification & intra-MCM PFN calculation****:* Each GPU chiplet can find which coalescing VPN is in its TLB by checking its LCF with candidate coalescing VPNs. The coalescing VPNs can be calculated by decrementing or incrementing the requested VPN by interlv_gran (Section IV-F and Example 4). To calculate addresses, F-Barre employs a PEC logic per GPU chiplet. The PFNs are calculated by using the same calculations in Section IV-F, Example 4. When IOMMU compiles an ATS response, if more than one bit is set in coal_bitmap in the PTE, it includes the 10-bit coalescing group information of the PTE and the 118-bit PEC buffer entry into the ATS response packet. The coalescing group information is added to the L2 TLB entry with the PFN. This increases the L2 TLB size by 1.3% and elongates the access latency by 0.5%, according to our CACTI

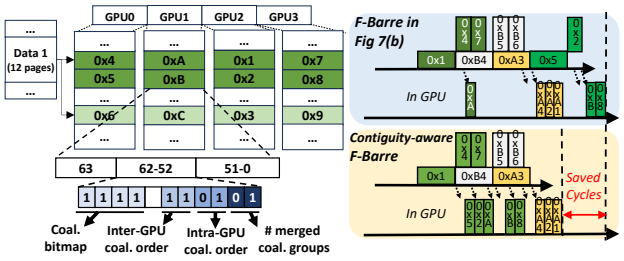


Fig. 13: PTE and Walkthrough for expanded coalescing groups

measurements. A PEC buffer information is maintained in a buffer in the PEC logic. A 5-entry PEC buffer (590 bits) takes 0.89% of L2 TLB size according to CACTI.

B. Contiguity-aware coalescing group expansion

In Barre, each coalescing group only covers one page per GPU chiplet. This makes the design simple and flexible without requiring any physical memory contiguity. But, when contiguous a few PFNs are available, Barre can opportunistically exploit the benefit of enlarging the coalescing group by adding two more parameters to the coalescing group information.

Fig 13 shows the extended PTE format and the walk-through example. Within the available 11 bits, we add *intra-GPU_coal_order* and *#_merged_coal_groups*, each means the position of the page in the coalescing group mapped on the same GPU chiplet and the number of merged coalescing groups. Simply put, intra/inter-GPU_coal_order parameters are (x, y) coordination of each page within the coalescing group. With this 2-dimensional coordination, the PFN can be estimated. The coalescing groups are expanded only when contiguous PFNs are commonly available in all GPU chiplets. As we opportunistically merge coalescing groups, we can support fragmented mappings as shown in Fig 14. The contiguity-aware F-Barre can map a 3-page data when there are not available consecutive three pages by allocating a two-page merged coalescing group and another one-page default-size coalescing group, while super pages cannot.

In the merged coalescing group, as in Barre, once one VPN is translated, all the other pages in the merged group can be calculated. With the translated PTE, the first VPN of the coalescing group is calculated as below.

$$VPN_{first} = VPN_{PTE} - \textit{intra-GPU\_coal\_order} - \textit{interlv\_gran} \times \textit{inter-GPU\_coal\_order}$$

As each GPU maps *#_merged_coal_groups+1* pages starting from (VPN_{first} + *interlv_gran* × *inter-GPU_coal_order*), inter-GPU_coal_order of the pending page can be estimated. As discussed in Section IV-F, inter-GPU_coal_order is used for finding the GPU id and the global base PFN that the page is mapped to. As LASP maps consecutive VPNs to each GPU chiplet, the intra-GPU_coal_order of the pending VPN is also estimated. With this information, the pending PFN is calculated below.

$$PFN_{pending} = PFN_{PTE} - base\_PFN_{PTE} - \textit{intra-GPU\_coal\_order}_{PTE} + base\_PFN_{pending} + \textit{intra-GPU\_coal\_order}_{pending}$$

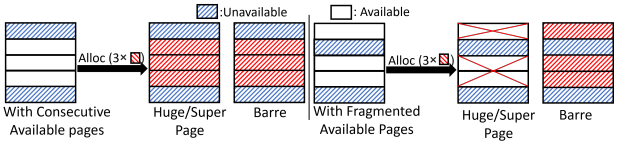


Fig. 14: Contiguity-aware F-Barre vs. Super page

C. Coalescing-aware PTW scheduling

In Barre, PTWs look up the pending coalescing requests after finishing a page table walk. While processing a page table walk, another PTW may fetch the pending coalescible requests and translate them with page table walks. We observe that notable coalescing opportunities are missed this way. Thus, we propose a lightweight *coalescing-aware PTW scheduling*. Every cycle, the scheduler checks if the request in the front of the queue is coalescible to any walking translations. If so, the scheduler de-prioritizes the request by putting it to the end of the queue such that the request can be fetched by a PTW that can do a coalesced PFN calculation. The coalescibility is examined by reusing the calculations in Section IV-F.

VI. Discussions

Page mapping: We use LASP in the baseline. Barre can support uneven page distribution by employing multiple interlv_gran fields per data and is also applicable to different page mapping. In Section VII-H6, Barre consistently shows speedup with CODA [21], round-robin [25], kernel-wide chunking [30]. This is because Barre exploits the unique GPU computing model where *all* CTAs use the same set of data. As far as data are mapped across multiple GPU chiplets and the accesses are not completely random, Barre can coalesce translations.

Security: Barre places pages in a coalescing group on the same local PFNs. This does not mean that PFNs can be estimated from VPNs. Pages will be mapped at run time. We leverage opportunistically available PFNs across GPUs. The coalescing groups can be mapped on different PFNs at every application instance.

Support for dynamically allocated data: Memory can be allocated during the kernel execution. Barre does not support these dynamically allocated data because these data tend to have limited access scope (within each CTA); there is not much chance to map the data across GPUs.

Support for on-demand paging & migration: Barre can be integrated with on-demand paging with minimal change. To maintain the coalescing group, pages will be fetched/evicted in the unit of coalescing groups. This is practical because the pages in the same coalescing groups tend to be accessed at similar times. Barre assumes that the pages in the coalescing group have a higher affinity to the mapped GPU chiplets according to LASP's locality analysis. Thus, there is little chance that the coalescing pages need to be migrated. But, if a page has high demand from remote GPU chiplets, we reset coal_bitmap to exclude the page from coalescing.

Scalability: Barre currently supports up to eight GPUs (up to four GPUs in contiguity-aware Barre Chord) due to

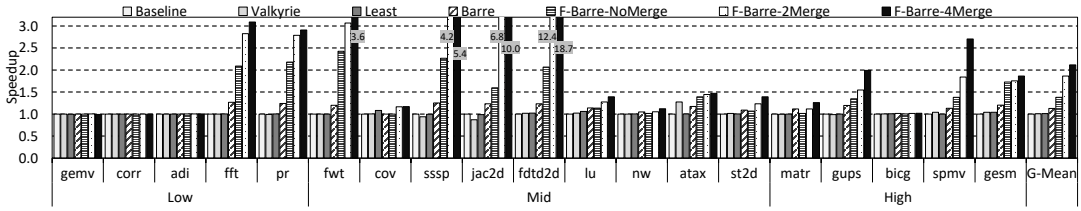


Fig. 15: Overall Performance Comparison

the limited available bits in a PTE. Barre can be adjusted to support more GPUs by configuring the coal_bitmap to use binary number representation rather than a bit map. For example, a bitmap `0110` means GPU chiplets 1 and 2. With the new configuration, it means consecutive six GPU chiplets in a coalescing group. We leave this as a design choice. We evaluated the performance impact with more GPUs in Section VII-H1.

DRAM address interleaving: For better rank, channel, and bank-level parallelism, commodity DRAMs use advanced address interleaving. Barre does not make any modification for this intra-DRAM address mapping. Once the same local PFN is determined for each coalescing group, the pages will be mapped across ranks, channels, and banks according to the individual DRAM controller's mapping algorithm.

TLB Shootdown: Barre correctly works when TLB shootdown. When IOMMU commands L1/L2 TLB shootdown, we reset all LCFs and RCFs as well such that any residue values in filters do not lead to mispredictions.

VII. Evaluation

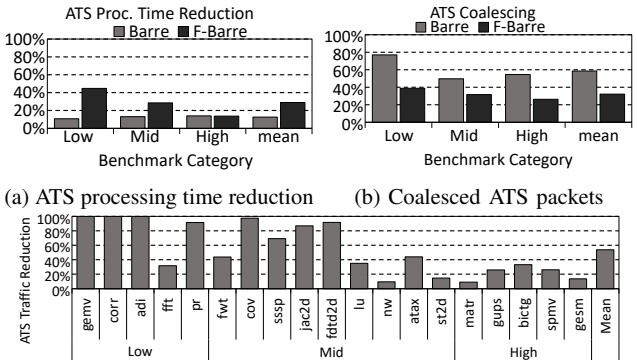
The basic settings are described in Section III-A and the detailed simulation parameters are listed in Table II. Throughout this section, we interchangeably use Barre Chord and F-Barre because F-Barre includes Barre.

A. Overall Performance

We compare the performance with two state-of-the-art solutions [8], [27]. Valkyrie [8] is extended with an inter-L1 TLB sharing and L2 TLB prefetch in our MCM-GPU. Least [27] is implemented by applying an ideal 1024-entry cuckoo filter (100% true positive) as the local TLB tracker.

TABLE II: Simulation Parameters

Parameter	Value
Number of GPU chiplets	4
Number of SAs	4 per Chip
Number of CUs	16 per SA. 256 in total
L1 Vector Cache	16 KB, 4-Way, 16 MSHRs
L1 Inst Cache	32 KB, 4-Way, 16 MSHRs
L1 Scalar Cache	16 KB, 4-Way, 16 MSHRs
L2 Cache	2 MB, 16-way, 64 MSHRs
DRAM	1 TBps, 100ns latency [41]
L1 TLB	64 entries, fully connected, 16 MSHRs, 1 cycle lookup latency, private to CU, LRU.
L2 TLB	512 entries, 16-Way, GPU chip-shared. 10 cycle lookup latency, 16 MSHRs.
IOMMU	16 shared PTWs. 500-cycle page table walks [27], [47], 48 PW-queue entries.
CTA/Page Scheduling	LASP [20]
Inter-chip bandwidth	768 GB/s mesh, 32 cycle latency [3], [51]
CPU-GPU Connection	PCIe Gen4 x16. 150 cycle latency [26], [27]
Cuckoo Filter	9-bit fingerprint, 4-way, 256 Rows (1024 entries) per filter
Merged Coalescing group	2 by default
PEC buffer	5 entries of 118 bits each



(a) ATS processing time reduction (b) Coalesced ATS packets

(c) ATS traffic reduction by F-Barre

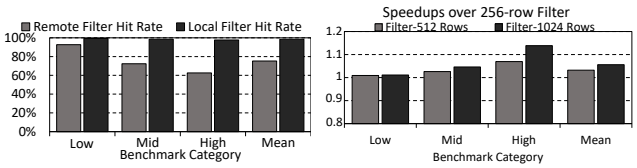
Fig. 16: Evaluations on ATS traffic

As shown in Fig 15, Barre achieves average of 10.05% to 12.8% speedups over Valkyrie and Least, respectively. F-Barre (F-Barre-NoMerge) further escalates the average speedups to 1.24× over Barre, which is 1.36× over Least. Though Valkyrie is effective in sharing translations across cores within a GPU, the performance gain of its local TLB probing is diminishing on multi-GPU platforms as the main performance bottleneck in multi-GPU is inter-GPU interference rather than intra-GPU locality. The inter-GPU TLB sharing in Least also has less impact because the opportunity to share exact TLB entries is reduced with advanced page mapping algorithms.

Impact of coalescing group expansion: We also evaluate the impact of contiguity-aware coalescing group expansion by allowing up to 2 and 4 merged coalescing groups (*F-Barre-2Merge* and *F-Barre-4Merge*). With 2 and 4 mergeable coalescing groups, F-Barre derives an average of 1.34× and 1.53× speedups over baseline F-Barre. Due to the limited available bits in PTE, we test up to four mergeable groups but as the performance scales well, we may use other unused bits to support coarser-grained merging. We will leave this as future study. We use F-Barre-2Merge as F-Barre for the rest of the evaluations.

B. ATS Traffic and Response Time

The main source of speedup is the improved ATS handling efficiency in IOMMU. Fig 16a presents the average ATS packet processing time reduction. Barre and F-Barre saves the ATS processing time by an average of 12.6% and 28%, respectively. As shown in Fig 16c, F-Barre cuts ATS packet traffic by on average 53% up to 99%, because significant amount of translations are handled **within** the MCM-GPU. `gemv`, `corr`, `adi` had almost 100% ATS traffic reduction



(a) Remote and Local Filter Hit rate

(b) Normalized speedup with diverse RCF and LCF size

Fig. 17: Evaluation on Filters

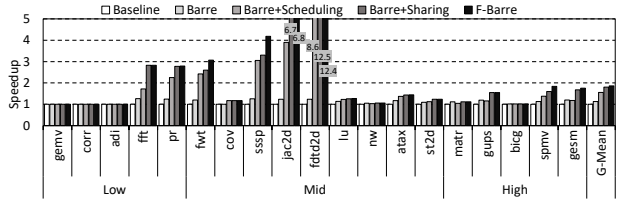


Fig. 18: Speedup breakdown

but they show negligible speedup in Fig 15 because they have low IOMMU intensity and hence the ATS handling is not the critical performance bottleneck. Contrarily, `fft`, `pr` in the low category show significant speedup because their IOMMU intensity is at least 10× higher than the other applications in that category as shown in Table I. Fig 16b plots the percentage of coalesced ATS packets in IOMMU. On average, Barre and F-Barre each coalesces 58% and 32% translations. F-Barre shows less coalescing rate because the majority of coalescing is happening within the MCM-GPU in F-Barre. Only unique translation requests are sent as ATS packets in F-Barre.

C. Peer PTE Presence Prediction Accuracy

For RCF and LCF evaluation, we define remote hit rate as the number of times when a peer GPU can provide the translation for the total peer translation requests and local filter hit rate as the true positive rate of the LCF. Fig 17a shows that the filters have 75.3% remote hit rate and a 98.4% local hit rate. RCFs show a lower hit rate because we use a best-effort strategy for the peer filter updates (i.e., no acknowledgment) and hence there can be missing updates.

D. Speedup Breakdown of F-Barre

Fig 18 shows the individual impacts of two optimizations of F-Barre. The coalescing-aware PTW scheduling provides 1.34× speedup over Barre by increasing the coalescing opportunities. The peer coalescing information sharing increases the performance to 1.80× over Barre. Because peer sharing removes PCIe traffic and implicitly provides enlarged TLB space, the performance gain is higher than PTW scheduling.

E. Traffic Overhead of Coalescing Information Sharing

To understand the performance impact of the peer-sharing traffic, we compare the F-Barre with an Oracle case where information is shared with a fixed latency (theoretical latency of inter-GPU communication) without exhausting bus or port resources. As shown in Fig 19, F-Barre achieves over 80% of the theoretical max performance. This means that F-Barre

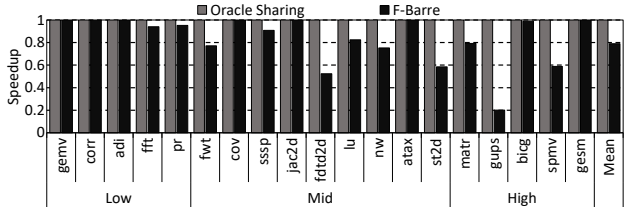


Fig. 19: Traffic overhead of coalescing information sharing

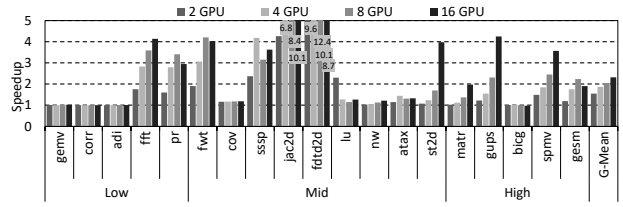


Fig. 20: Speedup with F-Barre on diverse-scale MCM-GPUs

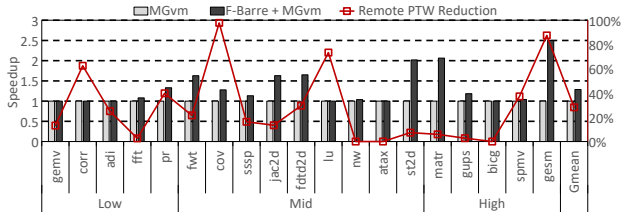


Fig. 21: Speedup on GMMU-integrated platform [41]

has room for further speedup by optimizing the peer-sharing traffic. We leave this as our future work.

F. Speedup on GMMU-integrated systems

In this section, we evaluate the impact of Barre Chord with GMMU. MGvm [41] is the state-of-the-art solution that uses private GMMU per GPU chiplet. To improve the locality of distributed GMMUs, MGvm extends LASP and uses a coarse-grained page mapping. Runtime monitoring is also used to fall back to fine-grained mapping when a few hot pages cause L2 TLB imbalance. We integrate Barre Chord to MGvm to evaluate the effectiveness on the GMMU-integrated platform. Fig 21 shows the performance of MGVm with and without Barre Chord. On average, Barre Chord further improves the performance of MGvm by 1.28×. This is because Barre Chord fundamentally removes local & remote page table walks and GMMU accesses while MGvm focuses on transforming remote page table walks to local ones. The red line graph in Fig 21 shows the reduced remote page table walks by Barre Chord. On average, Barre Chord can remove over 30% remote GPU chiplet accesses for page table walk than MGvm.

G. Speedup when page migration is enabled

Some GPUs support run-time migrations. NVIDIA GPUs monitor page access count and migrate the pages to the GPUs that have higher locality. Barre Chord also works under page migration. If some pages within a coalescing group have a higher locality with a certain GPU, the driver will migrate them according to the conventional GPU migration

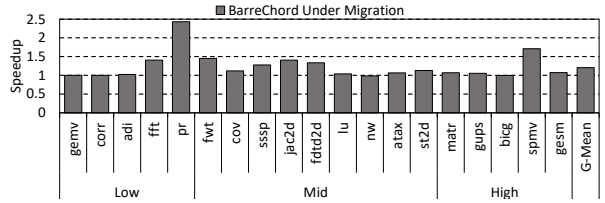


Fig. 22: Speedup with Barre Chord when migration is enabled

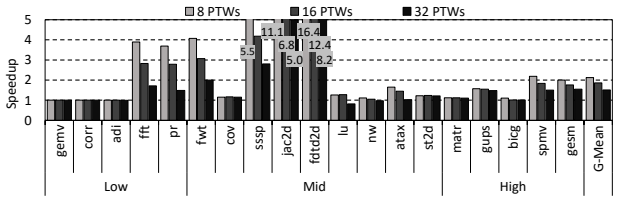


Fig. 23: Speedup with F-Barre with diverse PTW counts

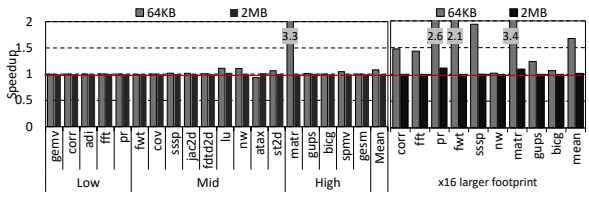


Fig. 24: Speedup of F-Barre when using large pages: (left) with original input size, (right) with 16× larger input

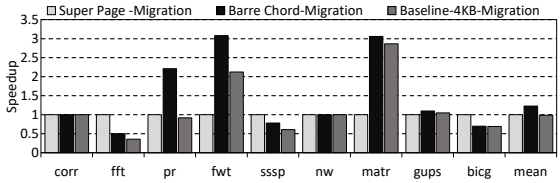


Fig. 25: Barre Chord vs. Super page

policy. By removing the pages from the coalescing groups, the pages no longer will be included in the Barre Chord address calculation without any penalty. The other pages in the coalescing group can still be translated via Barre Chord. To evaluate the performance impact of migration to Barre Chord, we implement the state-of-the-art counter-based page migration scheme, ACUD [7], which uses migration threshold as 16. Compared to ACUD, Barre Chord achieves an average of 1.20× speedup, as shown in Fig 22.

H. Sensitivity Study

*1) **Number of GPU chiplets in an MCM-GPU**:* We have described Barre Chord with four GPU chiplets. To understand the performance impact on diverse-scale MCM-GPUs, we evaluate F-Barre with 2-16 GPU chiplets by using the method proposed in Section VI-Scalability. Fig 20 shows that F-Barre derives 1.54×, 1.86×, 2.04×, and 2.31× speed-ups when using 2, 4, 8, and 16 GPU chiplets. `st2d`, `matr`, `gups`, and `spmv` show almost linear speedup on larger MCM-GPUs because F-Barre can mitigate the increasing contentions in PCIe and between PTWs in larger-scale MCM-GPUs.

*2) **Number of PTWs**:* We evaluate the speedup with F-Barre while varying the number of PTWs. As plotted in Fig 23, the average speedups are 2.12×, 1.86×, and 1.51× when using 8, 16, and 32 PTWs, respectively. Due to increased parallelism, the speedup of F-Barre is reduced with more PTWs, while F-Barre still provides notable performance gains over all tested configurations. This result proves that Barre Chord can be an alternative solution to improve performance especially when there is high pressure in IOMMU without increasing PTWs.

*3) **Cuckoo filter configuration**:* We evaluate the performance impact of different RCF and LCF size. Fig 17b shows the average speedup of 512- and 1024-row filters normalized by 256-row filters. On average, filters having 512 rows and 1024 rows can bring 3% and 6% speedups.

*4) **Page Size**:* Our baseline uses 4KB pages. We assess the performance of Barre Chord when using large pages. Fig 24 (left) shows the speedup of F-Barre when using 64KB and 2MB pages. Even with larger pages that inherently reduce the ATS traffic, Barre Chord derives further performance gain; an average of 2.5% and 0.12% speedup with 64KB and 2MB. The main reason for the limited speedup is the small memory footprint of the simulated workloads compared to the enlarged page size. To understand the impact by considering scalability, we use a similar approach in MGvm [41] by enlarging the application size by 16× (this reflects the scaled page size from 4KB to 64KB). Fig 24 (right) shows the performance of a subset of applications (a balanced number of workloads from each TLB MPKI class (low/mid/high).). On average, Barre Chord achieves 67% and 2% speedups with 64KB and 2MB pages, respectively.

*5) **Super page vs. Barre Chord**:* While the previous section shows the portability and speedup of Barre Chord over various page sizes, this section evaluates the effectiveness of Barre Chord and the super page (2MB) through head-to-head comparison. While the super page can enlarge the TLB reach, it transfers the burden to high page migration penalty and more remote memory accesses [5], [14], [33]. Upon migration, a larger chunk of data should be copied between two memory locations. If there are a few hot data, the entire large page-worth data should be migrated including those that do not need to be migrated, which leads to unnecessary ping-poing migrations. In contrast, Barre Chord does not compromise the migration size or require strict physical contiguity. Fig 25 shows the performance of the super page using 2MB pages and Barre Chord using 4KB pages when runtime migration is enabled. On average, Barre Chord achieves 1.22× speedup over the super page. For some applications such as `fft`, the super page provides better performance due to the linear access pattern. For some other applications that do not show a clear linear access pattern and have shared data across GPU chiplets (`pr` and `fwt`), Barre Chord provides more than 2× speedup.

*6) **Other page allocation schemes**:* While we use LASP in our baseline, Barre Chord is applicable to any other page mapping policies that map data across GPU chiplets. We evaluate Barre Chord with three page mapping policies, round-robin (used in [25]), CODA [21] and the kernel-wide

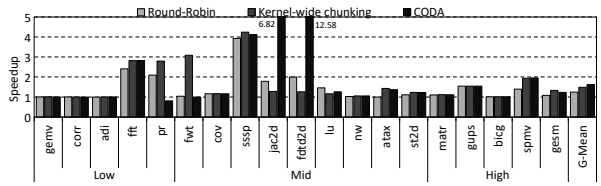
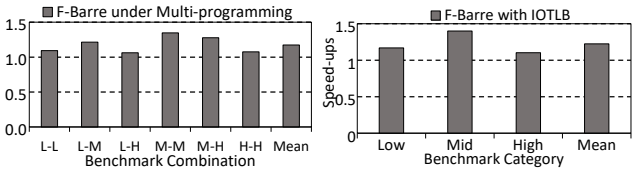


Fig. 26: Speedup under diverse page mapping



(a) Barre Chord under GPU multi-programming

(b) F-Barre with IOMMU TLB

Fig. 27: Impacts of different page mapping and IOMMU TLB

chunking [30]. CODA and chunking policies are described in Section II-B). Fig 26 shows the speedup of BarreChord. On average, Barre Chord provides 1.25×, 1.48× and 1.62× speedup with round-robin, kernel-wide chunking and CODA respectively, which demonstrates that Barre Chord is flexible to work with other page mapping policies. Barre Chord shows less speedup with locality-oblivious mapping policies such as round robin because the increased remote memory accesses dominate the performance overhead in some applications.

I. Multi-Application Support

Barre Chord considers the process ID and the access control associated to each page. Thus, Barre Chord can support multiple applications. To evaluate in a multi-programming environment, we randomly pick two applications having different IOMMU intensities (Table I) and run them concurrently on our baseline MCM-GPU. We apply fine-grained CTA-level sharing, where CTAs from both applications are co-located on any CUs by sharing on-chip resources [6], [46], [48]. Fig 27a shows the speedup with F-Barre. On average, F-Barre derives 17% speedup across all combinations. Middle-Middle combination shows the highest speedup at 34.7%. This is because the translation is not a critical bottleneck of Low-Low combination while intensive traffic in IOMMU hinder performance improvement for the High-High combination.

J. Combined with IOMMU TLB

The effectiveness of Barre Chord is evaluated by adding a 2048-entry IOMMU TLB having 200-cycle access latency [27] to the system. As shown in Fig 27b, F-Barre provides a further speedup by an average of 1.22× (up to 2.35×).

K. Hardware Overhead

In Barre, we add one PEC logic per PTW and one shared PEC buffer in the IOMMU. The PEC buffer has five 118-bit entries (total of 590 bits). Each PEC logic uses two comparators and a PFN calculator that is a small ALU with two subtractors/adders, and one multiplier. In F-Barre, we integrate four cuckoo filters (3 RCFs and 1 LCF), a PEC logic, and a PEC buffer per GPU chiplet. Each cuckoo filter has 256 rows and 4-way associativity, thus a total of 1024 9-bit fingerprints, which gives a 1.53% theoretical false positive rate. The total size of the four filers and a PEC buffer in one GPU chiplet is 4.57 KB, which takes 4.21% area overhead compared to a GPU L2 TLB according to CACTI [31] measurement.

VIII. Related Work

TLB Optimization: TLB performance optimization has been extensively studied on CPUs [1], [29], [34]–[37], [49], [50], GPUs [5], [18], [24], [26], [27], [39], [47], and accelerators [17], [23], [38]. Out of them, the most relevant works are TLB coalescing. Mosaic [5] arbitrates to use of a (coalesced) large page or a normal page in the unified virtual memory system based on applications' characteristics and targets to balance the trade-off of address translation and on-demand paging overhead. Snakebyte [24] opportunistically coalesces pages in a page group unit through runtime memory contiguity monitoring. Haria et al. [16] decouples the translation process and access control for heterogeneous systems and enforces strict linear mapping between virtual and physical addresses.

MCM-GPU & Multi-GPU Architecture Efficiency: Recently several studies explored efficient memory mapping and communication methods in MCM-GPUs and Multi-GPU architectures [3], [4], [20], [26]–[28], [41], [51]. Besides the studies [20], [27] already described earlier, Arunkumar et al. [3] propose the MCM design and explore different design spaces and optimizations, such as L1.5D cache, distributed CTA scheduling and first-touch page placement. MGvm [41] extends the locality type identified by [20] to TLB locality in a distributed page table scenario. FW-Trans [26] focuses on efficient page faults handling.

To the best of our knowledge, Barre Chord is the first study on the interplay between MCM-GPUs and IOMMU. We reveal that MCM-GPUs provide unique opportunities to coalesce page entries that enable novel calculation-based translation. Barre Chord is orthogonal to most existing contiguity-aware solutions as we do not require contiguous physical memory availability. Barre Chord shows superior performance than two most relevant state-of-the-art solutions [8], [27].

IX. Conclusion

Address translation is a critical performance bottleneck in an MCM-GPU. We propose Barre Chord that fundamentally reduces virtual memory translation overhead through translation coalescing, calculation-based translation, and intra-GPU translation. With a simple page mapping enforcement that is uniquely exploitable in MCM-GPUs, Barre Chord reduces the burden of expensive page table walks without sacrificing flexibility. Barre Chord shows superior performance compared to two state-of-the-art solutions.

Acknowledgement

This work was supported by NSF CCF-2114514. Part of this research was conducted using Pinnacles (NSF MRI, # 2019144) at the Cyber infrastructure and Research Technologies (CIRT) at University of California, Merced.

REFERENCES

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

[2] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and exploiting contiguity for fast memory virtualization," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 515–528.

[3] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 320–332.

[4] A. Arunkumar, E. Bolotin, D. Nellans, and C.-J. Wu, "Understanding the future of energy efficiency in multi-module gpus," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 519–532.

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

[6] A. Barnes, F. Shen, and T. G. Rogers, "Mitigating gpu core partitioning performance effects," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 530–542.

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

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

[9] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013, pp. 185–195.

[10] 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 (IISWC)*, 2009, pp. 44–54.

[11] N. Corp. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[12] 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 *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010, p. 63–74. [Online]. Available: https://doi.org/10.1145/1735688.1735702

[13] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *ACM International on Conference on Emerging Networking Experiments and Technologies*, 2014.

[14] K. Gosakan, J. Han, W. Kuszmaul, I. N. Mubarek, N. Mukherjee, K. Sriram, G. Tagliavini, E. West, M. A. Bender, A. Bhattacharjee, A. Conway, M. Farach-Colton, J. Gandhi, R. Johnson, S. Kannan, and D. E. Porter, "Mosaic pages: Big tlb reach with small pages," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 433–448. [Online]. Available: https://doi.org/10.1145/3582016.3582021

[15] GPUOpen-LibrariesAndSDKs, "Releases · gpuopen-librariesandsdks/ocl-sdk." [Online]. Available: https://github.com/GPUOpen-LibrariesAndSDKs/OCL-SDK/releases

[16] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 637–650. [Online]. Available: https://doi.org/10.1145/3173162.3173194

[17] B. Hyun, Y. Kwon, Y. Choi, J. Kim, and M. Rhu, "Neummu: Architectural support for efficient address translations in neural processing units," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1109–1124.

[18] A. Jaleel, E. Ebrahimi, and S. Duncan, "Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 1, pp. 1–24, 2019.

[19] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 international symposium on memory systems*, 2015, pp. 223–234.

[20] M. Khairy, V. Nikiforov, D. Nellans, and T. G. Rogers, "Locality-centric data and threadblock management for massive gpus," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1022–1036.

[21] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, "Coda: Enabling co-location of computation and data for multiple gpu systems," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, sep 2018. [Online]. Available: https://doi.org/10.1145/3232521

[22] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in gpus for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1357–1370.

[23] J. Landgraf, M. Giordano, E. Yoon, and C. J. Rossbach, "Reconfigurable virtual memory for fpga-driven i/o," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 556–571.

[24] 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 (HPCA)*, 2023, pp. 1195–1207.

[25] 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*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1163–1177. [Online]. Available: https://doi.org/10.1145/3613424.3614269

[26] 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 *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 456–470.

[27] B. Li, J. Yin, Y. Zhang, and X. Tang, "Improving address translation in multi-gpus via sharing and spilling aware tlb design," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1154–1168. [Online]. Available: https://doi.org/10.1145/3466752.3480083

[28] Y. Li, H. Qi, G. Lu, F. Jin, Y. Guo, and X. Lu, "Understanding hot interconnects with an extensive benchmark survey," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 3, p. 100074, 2022.

[29] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1023–1036.

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

[31] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[32] Ohio State University. [Online]. Available: https://sourceforge.net/projects/polybench/

[33] 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.

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

[35] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every walk's a hit: Making page walks single-access cache hits," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 128–141. [Online]. Available: https://doi.org/10.1145/3503222.3507718

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

[37] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 258–269.

[38] J. Picorel, D. Jevdjic, and B. Falsafi, "Near-memory address translation," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Ieee, 2017, pp. 303–317.

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

[40] 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 (HPCA)*, 2021, pp. 626–639.

[41] B. Pratheek, N. Jawalkar, and A. Basu, "Designing virtual memory system of mcm gpus," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 404–422.

[42] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 582–595.

[43] 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 *ISCA*. IEEE, 2018, pp. 180–192.

[44] 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 *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 197–209.

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

[46] X. S. Tan, P. Golikov, N. Vijaykumar, and G. Pekhimenko, "Gpupool: A holistic approach to fine-grained gpu sharing in the cloud," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2022, pp. 317–332.

[47] 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, pp. 191–204.

[48] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 230–242. [Online]. Available: https://doi.org/10.1109/ISCA.2016.29

[49] 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.

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

[51] S. Zhang, M. Naderan-Tahan, M. Jahre, and L. Eeckhout, "Sac: Sharing-aware caching in multi-chip gpus," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589078