Leveraging Chiplet-Locality for Efficient Memory Mapping in Multi-Chip Module GPUs

Junhyeok Park*
Electronics and Telecommunications Research Institute
Daejeon, Republic of Korea
vzx00770@skku.edu

Sungbin Jang
Sungkyunkwan University
Suwon, Republic of Korea
sunbi3361@skku.edu

Osang Kwon
Sungkyunkwan University
Suwon, Republic of Korea
osang915@skku.edu

Yongho Lee
Sungkyunkwan University
Suwon, Republic of Korea
jhyn205@skku.edu

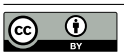
Seokin Hong
Sungkyunkwan University
Suwon, Republic of Korea
seokin@skku.edu

Abstract

While the multi-chip module (MCM) design allows GPUs to scale compute and memory capabilities through multi-chip integration, it introduces memory system non-uniformity, particularly when a thread accesses resources in remote chiplets. In this work, we investigate how page size in memory mapping affects this non-uniformity. Large pages reduce address translation overhead by covering larger memory regions per TLB entry; however, they enforce coarse-grained data placement, which can lead to data misallocation across chiplets. In contrast, small pages allow for finer-grained placement, increasing the likelihood of mapping data to the chiplet most likely to access it. We observe that application performance is sensitive to page size, with the appropriate configuration depending on workload characteristics.

This paper introduces *CLAP* which determines the suitable page size—specifically, how much data should be co-located within a single chiplet-for each application. We observe that GPU applications exhibit a distinct memory mapping pattern, in which specific groups of virtually adjacent pages are primarily accessed by the same chiplet with the group size tending to remain consistent - a property referred to as *chiplet-locality*. Leveraging this insight, CLAP predicts groups of pages exhibit chiplet-locality and pre-organizes them to contiguous physical frames within the chiplet most likely to access them. This organization forms regions that behave like large pages, as CLAP enables these page groups to be covered by a single merged TLB entry through deliberate virtual-to-physical contiguity. As a result, CLAP delivers the benefits of large pages without compromising chiplet-level memory locality. Our evaluation shows that CLAP improves performance by up to 19.2% compared to previous paging schemes.

*This work was done while the author was at Sungkyunkwan University.



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756090

CCS Concepts

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

Keywords

GPUs, Multi-Chip Module (MCM), Virtual Memory, Page Placement

ACM Reference Format:
Junhyeok Park, Sungbin Jang, Osang Kwon, Yongho Lee, and Seokin Hong. 2025. Leveraging Chiplet-Locality for Efficient Memory Mapping in Multi-Chip Module GPUs. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3725843.3756090

1 Introduction

GPUs have steadily scaled their compute and memory capabilities to meet growing performance demands. However, as transistor scaling approaches physical limits and die sizes plateau ($\sim 800 mm^2$) [13, 14, 39], scaling monolithic GPU designs becomes increasingly challenging. In response, multi-chip module (MCM) designs, which integrate multiple chiplets within a single package, have emerged as a promising solution and are already being adopted by major GPU vendors [4, 6, 76, 77, 95].

MCM designs, while enabling improved scalability, introduce a critical challenge: memory non-uniformity. Accessing data on remote chiplets incurs additional latency and energy consumption, thereby degrading overall system performance [13, 14]. Prior research has addressed this challenge through a variety of techniques [13, 27, 47, 64, 87, 90, 102, 106, 109, 111, 112], including (1) GPU thread and data placement optimization [13, 47, 49, 102], (2) remote data caching into local resources [27, 64, 90, 106, 109, 111, 112], and (3) improvements to address translation paths [32, 87]. However, the impact of *page size*, a key factor in memory mapping, remains largely unexplored.

Why is Page Size Important?: Page size plays a critical role in the memory systems, particularly in address translation. Larger pages reduce address translation overhead by allowing a single TLB (Translation Lookaside Buffers) entry to cover a wider virtual address range [36, 53, 82, 83, 89]. In our evaluation, 64KB and 2MB pages reduce average address translation latency by 17.1% and 48.7%, respectively, compared to 4KB pages. Consequently, modern GPUs actively support large pages to leverage these benefits [72, 74].

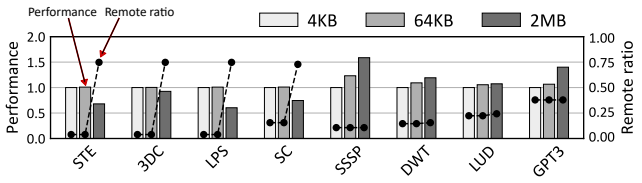


Figure 1: Performance (bar, normalized to the 4KB page case) and remote access ratio of memory instructions (line) across various page sizes. The results demonstrate that each application exhibits a distinct sensitivity to page size.

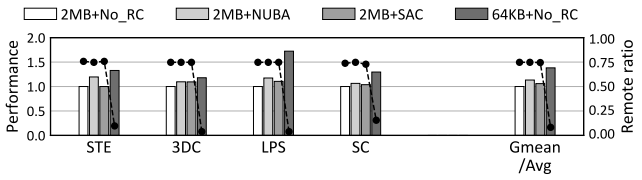


Figure 2: Performance (bar) and remote access ratio of memory instructions (line) for different page sizes and remote caching schemes. 'No_RC' denotes 'without remote caching.' The results demonstrate that, even with remote caching, selecting an appropriate page size remains critical.

For instance, in NVIDIA GPUs, memory allocations made through `cudaMalloc()` are backed by 2MB large pages [69, 74].

However, in MCM designs, page size also influences memory non-uniformity. A larger page maps a contiguous virtual address region to a corresponding contiguous physical frame located within a single chiplet. This coarse-grained placement can lead to data misplacement when portions of a page are accessed by threads executing on other chiplets. Figure 1 shows the impact of different page sizes on overall system performance. For workloads such as STE, 3DC, LPS, and SC, increasing page size degrades performance due to increased remote accesses. In contrast, SSSP, DWT, LUD, and GPT3 benefit from larger pages, as they reduce address translation overhead without incurring additional remote access. These results indicate that judicious page size selection is essential to improving system efficiency and performance within MCM designs.

One might expect that using large pages and caching remote data into local resources (i.e., remote caching) could mitigate this challenge. To evaluate this, we compare the performance across four configurations: (1) 2MB paging without remote caching; (2) 2MB paging with two state-of-the-art remote caching schemes-NUBA [111] and SAC [109]; and (3) 64KB paging without remote caching. As shown in Figure 2, the remote caching schemes moderately alleviate memory non-uniformity for benchmarks with high remote access ratios under the 2MB page size, achieving average speedups of 13.1% and 5.8%, respectively. In contrast, reducing the page size to 64KB proves more effective, delivering a 36.7% average speedup by significantly reducing remote memory accesses. This disparity arises from excessive remote memory accesses caused by inappropriate page sizes, which overwhelm the capacity of remote caching and limit the effectiveness of prior techniques. These findings underscore that appropriate page size selection remains fundamental to achieving high performance in MCM GPUs, even when prior performance optimization techniques are employed.

Memory Mapping with Suitable Page Sizes: This paper introduces *CLAP* (Chiplet-Locality Aware Page Placement), a mechanism for efficient memory mapping in MCM GPUs that exploits a unique characteristic of GPU workloads: *chiplet-locality*. Chiplet-locality refers to the tendency for specific groups of virtually contiguous pages to be primarily accessed by the same chiplet, with the group granularity remaining consistent within a data structure. This behavior stems from the parallelism of GPU programming, making data access patterns across chiplets more predictable.

CLAP first profiles the memory mapping of data structures to identify their chiplet-locality degree. To this end, it preemptively maps a subset of each data structure's pages to chiplets using a locality-based policy, creating a sample mapping. CLAP then monitors which chiplets access these pages and into which chiplets the pages are mapped. Based on this analysis, it selects an appropriate page size (i.e., the granularity of contiguous pages) and applies it by mapping virtually contiguous pages to contiguous physical frames of the chosen size within the chiplet predicted to access them. These regions effectively act as large pages, as CLAP leverages the constructed page contiguity to generate merged TLB entries and extend TLB reach. As a result, CLAP achieves both efficient address translation and high intra-chiplet data locality, which align with the characteristics we define as the suitable page size.

With accurate chiplet-locality analysis, CLAP can preemptively organize memory mappings with appropriate sizes, thereby avoiding page remapping via inter-chiplet migrations and eliminating their associated overhead. CLAP is fundamentally orthogonal to existing NUMA-addressing techniques [64, 90, 106, 109, 111], and can be synergistically integrated with them to further enhance their effectiveness. Our evaluation shows that CLAP improves performance by up to 19.2% over prior paging schemes [28, 32, 34, 87, 104].

Overall, this paper makes the following contributions:

- We analyze the impact of page size in MCM GPUs, identifying a key trade-off between address translation efficiency and memory access locality. Our study shows that GPU workloads exhibit diverse memory access characteristics, even across data structures within the same workload, resulting in different page-size preferences. Therefore, selecting an application-appropriate page size is critical to achieving high performance in MCM designs.
- We introduce *chiplet-locality*, a distinctive characteristic of GPU workloads that emerges in MCM designs. Chiplet-locality captures how GPU data structures are accessed across chiplets, revealing groups of pages that exhibit similar access patterns. This property provides key insight into determining appropriate data-placement granularity and policies.
- We propose *CLAP*, a mechanism for efficient page mapping in MCM GPUs. CLAP ensures that groups of pages exhibiting chiplet-locality are mapped contiguously to the appropriate chiplet, thereby constructing large page–like regions. This design enables the benefits of large pages while preserving memory locality within chiplets.

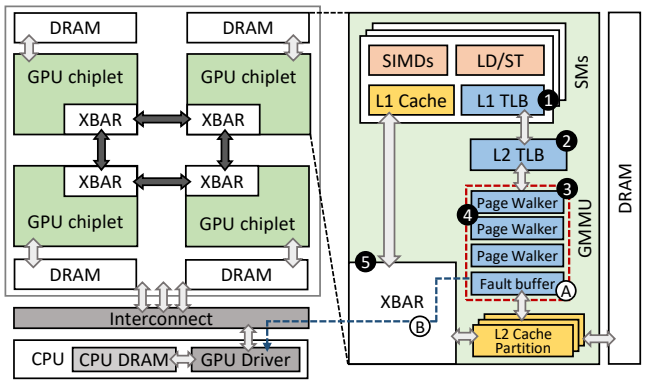


Figure 3: The baseline MCM GPU architecture.

2 Background

2.1 MCM GPU Architecture

Figure 3 illustrates an MCM GPU consisting of four GPU chiplets. Each chiplet contains multiple Streaming Multiprocessors (SMs) [1] and memory partitions. The chiplets are interconnected via on-package interconnects, enabling global sharing of memory partitions across all chiplets. Within each chiplet, crossbar switches (Xbars) manage memory accesses by routing them to either local or remote memory partitions. This architecture introduces NUMA effects, where remote memory accesses incur higher latency, consume off-chip bandwidth, and increase energy consumption[13, 14].

2.2 NUMA: Still Relevant?

In recent years, processor vendors have sought to mitigate NUMA (Non-Uniform Memory Access) effects in MCM designs by introducing dedicated I/O dies for memory integration and incorporating additional cache layers [6, 8, 95]. In parallel, advancements in interconnect technology have improved off-chip memory bandwidth while reducing energy consumption and access latency [78, 79, 95, 97].

Despite these advancements, NUMA effects cannot be fully eliminated due to fundamental physical constraints. For instance, in NVIDIA H100 GPUs [44, 73]—which consist of two logical memory partition groups—global memory access latency ranges from approximately 550 to 730 cycles, depending on the physical proximity between compute cores and memory partitions.[2] This latency disparity is expected to worsen in larger-scale MCM designs, as increased chiplet-to-chiplet communication cost further exacerbates memory non-uniformity. This trend indicates that addressing NUMA effects is essential for sustaining performance scalability in emerging MCM architectures.

2.3 Difference from prior NUMA GPUs

While MCM GPUs fall under the broader category of NUMA GPUs, they fundamentally differ in three key aspects.

First, while both prior NUMA GPUs (e.g., multi-GPU systems) and MCM GPUs require and benefit from explicit workload distribution (i.e., threads and data) and consistency maintenance, their architectural approaches differ. NUMA GPUs consist of independent devices, while MCM GPUs operate as a monolithic system that abstracts hardware-level complexity from both the OS and the programmer [13, 95]. This abstraction reduces opportunities for programmer-driven optimization and instead demands more robust system-level support.

Second, unlike NUMA GPUs with separate physical address spaces and per-device GPU page tables, MCM GPUs employ a unified address space and a single GPU page table shared across all chiplets. As a result, techniques such as page duplication [65, 104] become inapplicable, as a page table prohibits mapping a single virtual address to multiple physical locations.

Lastly, because MCM GPUs integrate chiplets at the package level, rather than at the board or system level, they present distinct performance trade-offs. While NUMA GPUs often benefit from page migration to mitigate the high cost of remote memory accesses, MCM designs may instead prefer remote access due to the relatively low overhead of intra-package communication [13, 101]. These fundamental differences highlight the need for architectural strategies tailored to the characteristics of MCM designs.

2.4 Virtual Memory System Design

An MCM GPU maintains a single GPU page table to support a unified physical address space. Not only application data structures but also the page table entries (PTEs) themselves are managed at page granularity (typically 4KB). As a result, PTE pages can be distributed across memory partitions in different chiplets, depending on the PTE page mapping policy [1, 81, 87, 88].

In the TLB hierarchy, the L1 TLB is located per SM, while the L2 TLB is either shared across SMs within each chiplet (i.e., chiplet-private) or across all SMs in the GPU (i.e., chiplet-shared). Our baseline adopts the chiplet-private L2 TLB configuration, in which memory requests access only the TLBs within their originating chiplet. Each chiplet is equipped with a GPU Memory Management Unit (GMMU), which handles TLB misses using hardware page table walkers. The page table walker traverses the multi-level in-memory page table [93, 94]; at each level, the page table access can be either local or remote, depending on the physical location of the corresponding entry [1, 81, 87, 88].

2.5 Memory Access Flow in MCM GPUs

Figure 3 outlines the memory access flow in an MCM design. When SIMD units initiate a memory access request, it first consults the multi-level TLBs (❶, ❷) for virtual-to-physical address translation. On a TLB miss, the request is forwarded to the local GMMU (❸). Multi-threaded page walkers then process the request by performing a page walk (❹). Once address translation is complete, the request retrieves data from the physical address. Depending on the location of the data, the request is routed either to a local memory partition or to a remote chiplet via the crossbar interconnect (❺).

In Case of Demand Paging: When a system employs a demand paging scheme (similar to unified virtual memory, UVM [3, 11, 91]), a data page may be absent from the GPU memory, causing a page fault. This fault is detected during the page walk process. If the page walk fails, it triggers a page fault, which is logged in the GMMU's Fault Buffer (Ⓐ). The GMMU then notifies the host GPU driver

[1]The term *Compute Unit (CU)* is used in AMD terminology. For consistency, we adopt NVIDIA terminology throughout this paper.
[2]Estimated on commodity GPUs using a microbenchmark.

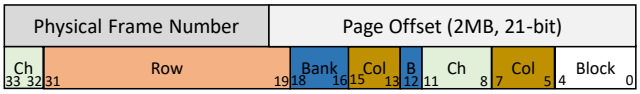


Figure 4: Memory interleaving policy in MCM designs. The two MSBs of the channel bits are outside of the page offset and serve as a chiplet identifier.

of the fault via a shared on-package I/O module. The GPU driver resolves the fault by migrating the faulted page from CPU memory to GPU memory and updating the GPU page table (Ⓑ). Once the fault is resolved, the GPU resumes the address translation process.

2.6 NUMA-Aware Memory Interleaving

Conventional GPUs enhance channel-level parallelism by interleaving data across multiple memory channels at sub-page granularity (i.e., 256B). However, in MCM designs, it limits the GPU driver's ability to optimize data placement in a NUMA-aware manner. To mitigate these issues, interleaving must be designed to avoid interfering with page-level locality, allowing the GPU driver to assign entire pages to specific chiplets [111].

Figure 4 illustrates a memory interleaving policy for a 34-bit physical address in a 4-chiplet MCM GPU with 64 memory channels (6 channel bits). In this policy, the two most significant bits (MSBs) of the channel bits are placed outside the page offset (2MB in this example) and serve as chiplet identifiers. This approach enables the GPU driver to enforce intended data placement while preserving channel-level parallelism within each chiplet. It is worth noting that a similar policy is already supported for NUMA-optimized workloads in recent chiplet-based designs [5, 7–9].

We observe that even without any NUMA optimization, this interleaving policy does not lead to any noticeable performance degradation, showing only a 0.6% difference compared to the naive interleaving used in monolithic GPUs. Furthermore, with a simple NUMA-aware optimization (used as the baseline in this paper) [13], it outperforms the naive configuration by 42.0%.

2.7 Threadblock and Data Arrangement

Threadblock (TB) and data arrangement are particularly important in MCM designs, as inaccurate placement can exacerbate memory non-uniformity [13, 47]. We categorize two representative approaches from prior work.

The first approach, referred to as **First-Touch-based (FT)**, schedules contiguous threadblocks within the same chiplet and places data in the chiplet where the first-touching thread resides [13]. This policy ensures that adjacent threadblocks exploit spatial locality, while data is placed close to the threads accessing it. The second approach, referred to as **Static-Analysis-based (SA)**, leverages compiler-assisted static analysis to determine threadblock-data locality [47]. This method analyzes program code to identify locality patterns and arranges threadblocks and data according to predefined placement policies.

3 Motivational Study

In this section, we explore (1) the impact of page size in MCM designs and (2) chiplet-locality in GPU workloads. We first describe our system setup and then delve into each key observation.

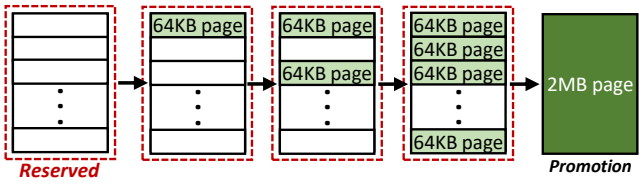


Figure 5: Paging with physical frame reservation.

3.1 System Configuration

Threadblock and Data Arrangement: We adopt the widely used FT policy [13, 27, 64, 90, 106, 108, 109] as our baseline, as it effectively exploits data locality for both regular and irregular applications. Although FT may incur paging overhead due to demand paging [113], this cost is fundamentally a one-time data transfer, which is also incurred in other schemes during their initial data placement.[3]

Virtual Memory Support: Our baseline system adopts *demand paging with physical frame reservation*, similar to current GPU unified virtual memory systems [3, 74, 85, 91] and prior paging strategies [68, 107]. Figure 5 illustrates an example of paging with physical frame reservation. For a large page (e.g., 2MB), the GPU driver first reserves a free physical frame of the corresponding size. Subpages (e.g., 64KB) are then mapped to the reserved frame on demand. Once all subpages are populated, the GPU driver promotes the region to a large page. This strategy provides a consistent demand paging granularity across page sizes and maintains uniform hardware resource usage (e.g., CPU-GPU interconnect bandwidth), regardless of page size. Our baseline supports three page sizes-4KB, 64KB, and 2MB-which are commonly used in modern GPU systems [72, 74]. We exclude larger page sizes (e.g., 512MB and 1GB) as they are considered too coarse-grained.

3.2 Methodology

We extend GPGPU-Sim v4.2 [48] to model a 4-chiplet MCM GPU. Our model is based on prior research and the specifications of

[3]We also discuss an alternative threadblock and data arrangement strategy—SA policy [17, 47]—as well as a different programming model (i.e., the conventional copy-and-execute model) in Section 5.2.

Table 1: Baseline simulation configuration.

Chiplets	4
GPU core	64 SMs per chiplet, 256 SMs in total, 1132MHz Max 64 warps per SM
L1 cache	128KB, 20-cycle, 128B line, per-SM
L2 cache	4MB per chiplet, 16MB in total, 160-cycle, 128B line SM-side cache, software-managed cache coherence [13]
L1 TLB	32-entry (4KB), 16-entry (64KB), 8-entry (2MB) 10-cycle, fully-associative, per-SM
L2 TLB	1024-entry (4KB, per chiplet), 512-entry (64KB, per chiplet) 256-entry (2MB, per chiplet) 80-cycle, 8-way, chiplet-private
Inter-chip BW	768GB/s per GPU, Ring topology, 32ns latency [13]
DRAM	HBM2 [23], 877MHz, 8 memory stacks, 8 channels/stack 16 channels per chiplet, 1.8TB/s in total tRCD = 14, tRP = 14, tCL = 14, tWL = 2, tRTP = 3
Page table	4-level page table, page size support: 4KB, 64KB, and 2MB
GMMU	256-entry page walk queue (per chiplet) 128-entry page walk cache (per chiplet) 16 page walkers (per chiplet)
TB & data arrangement	FT-based (FT) [13]

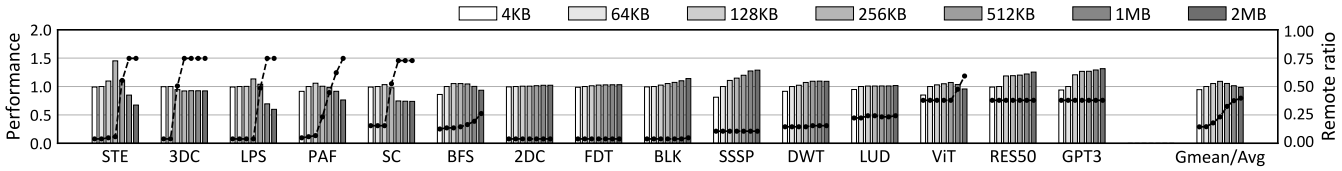


Figure 6: Performance (bar, normalized to the 64KB page case) and remote access ratio of memory instructions (line) for various page sizes. Each application achieves improved performance with its preferred page size (typically larger but with a lower remote access ratio), highlighting the variability in sensitivity to page size.

Abbr.	Workload	Input Size	Number of TBs	L2$ MPKI (4K/64K/2M)	L2 TLB MPKI (4K/64K/2M)
STE	stencil [96]	128MB	1024	6.65/6.83/11.7	1.63/1.38/0.55
3DC	3d convolution [35]	512MB	256	4.45/4.47/7.46	1.01/0.96/0.95
LPS	laplace3d [18]	1.00GB	2048	4.64/4.83/8.14	1.85/1.37/1.35
PAF	pathfinder [25]	1.87GB	1158	2.35/2.42/4.10	0.68/0.66/0.59
SC	streamcluster [25]	2.02GB	256	16.2/16.2/28.0	4.42/4.41/4.21
BFS	breadth-first-search [22]	242MB	6116	8.74/8.79/9.94	1.47/0.92/0.54
2DC	2d convolution [35]	512MB	262144	6.31/6.32/6.39	3.14/2.27/2.02
FDT	fdtd2d [35]	3.01GB	1048576	7.27/7.29/7.31	3.47/3.32/2.88
BLK	blackscholes [75]	310MB	62500	8.54/8.59/8.60	1.97/1.97/0.73
SSSP	single source shortest path [24]	1.79GB	374178	18.6/18.7/18.8	8.31/6.27/3.57
DWT	2d dwt [25]	496MB	65536	10.4/10.5/10.5	2.37/1.58/1.06
LUD	lud [25]	4.00GB	65536	1.90/1.95/2.00	0.89/0.60/0.10
ViT	GEMM [75], ViT-FC [29]	60MB	8192	1.48/1.50/1.52	0.66/0.46/0.28
	Shape: 8192 × 1024 × 768 (M × N × K)				
RES50	GEMM [75], ResNet50-FC [38]	104MB	8192	1.23/1.32/1.37	0.54/0.52/0.18
	Shape: 8192 × 1024 × 2048 (M × N × K)				
GPT3	GEMM [75], GPT3-FC [21]	2.31GB	24992	0.93/0.93/0.98	0.24/0.23/0.02
	Shape: 64 × 5000 × 12288 (M × N × K)				

Table 2: Evaluated workloads.

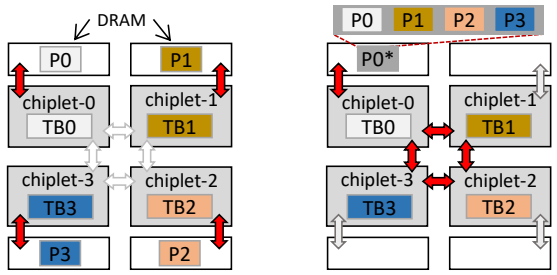
commercial GPUs [13, 70], parameters are detailed in Table 1. To support multiple page sizes, we allocate a separate TLB for each page size. PTE pages are distributed across chiplets, as proposed in prior work [87], to reduce remote accesses during page walks.

We evaluate 15 workloads from diverse benchmark suites [22, 25, 35, 67, 75, 96], selecting those with sufficient parallelism to utilize a 4-chiplet GPU system. We also include several matrix operations extracted from machine learning (ML) pipelines, using matrix dimensions from fully connected layers of various ML models [21, 29, 38]. These matrix operations cover a range of shapes and exhibit distinct memory access characteristics. Table 2 summarizes the characteristics of the workloads, including total input size, the number of threadblocks in the dominant kernel, and the L2 cache and L2 TLB MPKI (misses per kilo warp instructions) under memory mappings with 4KB, 64KB, and 2MB pages.

3.3 Impact of Page Size in MCM Designs

Figure 7 illustrates a scenario where each threadblock (TB0–TB3), scheduled on a different chiplet, accesses an exclusive data region (P0–P3). In a small page configuration (Figure 7a), each data region (P0–P3) is divided into separate pages, allowing the GPU driver to place each page near the chiplet requesting the data.

However, this locality is compromised when large pages are used. As shown in Figure 7b, a large page P0* spans the contiguous regions P0–P3, assuming that the large page is four times larger than a small page. When TB0 in chiplet-0 first accesses P0, the GPU driver places the entire large page P0* in chiplet-0. Consequently, TB1–TB3 subsequently access their respective regions (P1–P3), which now reside in chiplet-0, leading to remote accesses.



(a) Data placement with small pages.

(b) Data placement with large pages.

Figure 7: Data placement with different page sizes. Utilizing large pages can increase memory non-uniformity.

We analyze the impact of various page sizes on GPU workloads, including standard sizes (4KB, 64KB, and 2MB) and hypothetical sizes between 64KB and 2MB. Intermediate sizes between 4KB and 64KB are excluded due to their negligible impact in that range. To accurately model the performance impact of hypothetical page sizes, we add extra TLBs for each size (16 entries for L1 and 512 entries for L2). Figure 6 shows that workloads exhibit varying sensitivities to page size. Workloads on the left experience a high remote access ratio as the page size approaches 2MB, leading to performance degradation despite reduced address translation overhead. In contrast, workloads on the right maintain a stable remote access ratio across page sizes and show improved performance as the page size increases. This finding underscores the importance of adaptive page size selection in MCM systems.

Challenges: However, determining the suitable page size for diverse workloads is challenging. First, the suitable page size does not necessarily correlate with the total input size or the size of individual data structures, where a data structure refers to a GPU memory allocation (e.g., a call to `cudaMalloc()` or `cudaMallocManaged()`). For instance, PAF, which exhibits an irregular memory access pattern, achieves the best performance with a 128KB page size, even

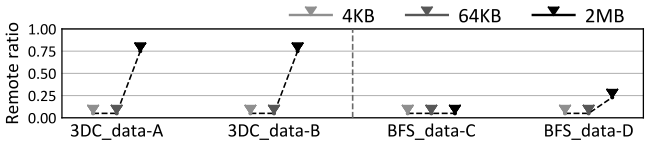


Figure 8: Remote access ratio of memory instructions (line) for data structures under various page sizes. Even within a single workload (e.g., BFS), different data structures can exhibit varying sensitivities to page size.

though its total input is nearly 2 GB and its largest data structure is 1.86 GB. In contrast, BLK, with regular memory access patterns, prefers a 2MB page size, despite having a total input of only 310 MB and individual data structures averaging just 77 MB. Second, workloads, such as STE, LPS, and SC, achieve higher performance with hypothetical page sizes that are not supported in current systems. Finally, even within a single workload, individual data structures can prefer different page sizes due to their different characteristics. Figure 8 shows the remote access ratio of two representative data structures in 3DC and BFS under various page sizes. In 3DC, data structures show consistent sensitivity to page size changes, following similar trends, while in BFS, their sensitivity varies.

3.4 Chiplet-Locality in GPU Workloads

Chiplet-locality refers to the tendency of particular groups of virtually contiguous data pages to be primarily accessed by threads on the same chiplet, with the group granularity remaining consistent within a data structure. Figure 9 illustrates an example of chiplet-locality in a GPU data structure. The data structure is distributed across different chiplets, with the GPU driver mapping each data page to the chiplet that requests access to it. The page table shows the mapping between virtual pages and physical frames. Notably, four virtually contiguous pages are accessed by, and mapped to, the same chiplet (e.g., VPN 0x100–0x103), and this pattern is repeated across other segments (e.g., VPN 0x104–0x107).

We observe that chiplet-locality is a prevalent property of GPU workloads. To quantify this, we measure the proportion of each data structure's address range that exhibits chiplet-locality across diverse GPU workloads. Our analysis covers nearly all data structures in each workload, calculating the proportion per structure and then averaging across them (the group granularity may vary between structures). Data structures smaller than 2MB are excluded, as they typically have low access frequency and negligible impact on overall performance. For GEMM operations, one of the three matrices (i.e., matrix B in $C = A \times B$) is accessed by all threads in the kernel, resulting in a global access pattern. We classify this case as 100% chiplet-locality, since from the perspective of each chiplet, the entire address space of the matrix is uniformly accessed. Each structure is mapped using small pages (i.e., 64KB) to enable fine-grained placement of each page onto its desired chiplet. Figure 10 shows that GPU data structures exhibit high chiplet-locality, with an average

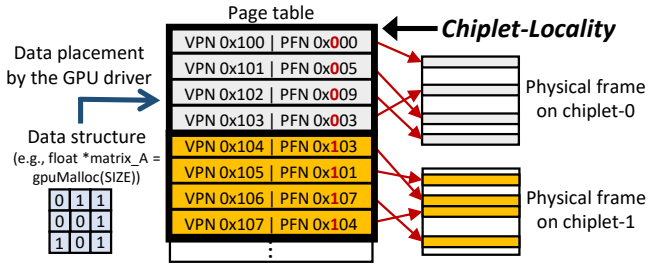


Figure 9: An example of chiplet-locality in a GPU data structure. The four contiguous pages (VPNs 0x100–0x103) are predominantly accessed by chiplet-0, and this pattern appears consistently across other regions (e.g., VPNs 0x104–0x107).

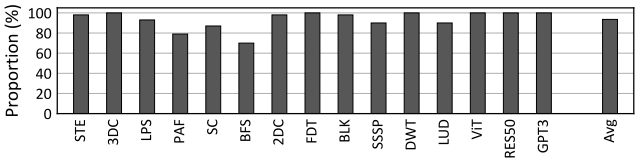


Figure 10: Proportion of the address range of GPU data structures exhibiting chiplet-locality. The data structures typically highly exhibit chiplet-locality.

of 93.5% of the memory range demonstrating this behavior. This arises from the intrinsic characteristics of GPU workloads: threads within a kernel execute the same program code, leading to similar memory access patterns. In particular, adjacent threads (e.g., within a threadblock located in a chiplet) tend to access spatially adjacent memory regions [37, 51, 80, 103].

3.5 Revisiting Prior NUMA Solutions

In MCM-GPUs, numerous studies have addressed the NUMA challenge [13, 32, 47, 87, 109, 111], but most focus primarily on one of two key aspects of memory: physical data access, which refers to fetching data from caches or GPU DRAM after address translation, or the address translation itself.

First, various threadblock and data arrangement strategies [13, 47, 49] improve physical data access by enhancing data locality within chiplets. Similarly, remote data caching schemes [109, 111] further enhance it by storing remotely mapped data in local resources. However, these approaches do not thoroughly consider how virtual memory configurations, such as page size and address translation, affect their effectiveness.

Other schemes aim to reduce address translation overhead by applying NUMA-aware optimizations in GPU page table and TLB entry placement, as well as page walk scheduling [32, 87]. While these approaches also consider physical data access, they primarily extend existing NUMA optimizations [47]. As a result, they assume favorable locality and do not fully consider how translation and data access interact under different virtual memory configurations.

In NUMA-CPU systems, prior work has sought to optimize both physical data access and address translation by dynamically constructing and splitting large pages at runtime via page migrations—an approach referred to as **C-NUMA** [28, 34]. Despite its benefits, this approach has four major limitations.

First, C-NUMA does not consider the distinct characteristics of individual data structures within an application. As a result, it can exhibit suboptimal or incorrect behavior.

Second, it heavily relies on page migration, which incurs significant overhead. Migrating pages within or across chiplets requires costly operations such as TLB shootdowns and cache flushes. These operations introduce additional latency and degrade the efficiency of both physical data access and address translation, reducing overall GPU performance [12, 100].

Third, it provides limited flexibility, supporting only a few specific page sizes (e.g., 4KB and 2MB). Consequently, it cannot accommodate workloads that benefit from intermediate page sizes. Even if such sizes were supported, it lacks a mechanism to determine the suitable page size for a given application.

Technique	GPU-friendly?	Consideration		
		A: Physical data access	**B:** Virtual memory	**C:** Interaction between A & B
TB & data arrangement [13, 47]	✓	✓	△	✗
Remote data caching [109, 111]	✓	✓	△	✗
Reducing translation overhead [32, 87]	✓	△	✓	✗
C-NUMA [28, 34]	✗	△	△	✓
Our approach	✓	✓	✓	✓

Table 3: Comparison with prior NUMA solutions.

Lastly, it adopts a reactive strategy, adapting its behavior to the application's current execution state. While this is effective in CPU systems—where workloads are unpredictable and processes can be rescheduled across chiplets, changing their memory access patterns—it is not well-suited for GPUs. GPU schedulers do not migrate threads across cores, since thread migration requires relaunching threads, which incurs substantial overhead. Furthermore, as discussed in Section 3.4, GPU workloads exhibit stable and predictable access patterns driven by massive parallelism, making C-NUMA's reactive approach ill-suited to GPUs. Therefore, the design direction for more advanced solutions must be re-evaluated.

Our Approach: In this paper, we propose a new solution to improve MCM GPU performance by jointly optimizing physical data access and virtual memory management. Specifically, our design: (1) rapidly profiles application behavior through a low-overhead mechanism, (2) adapts to the distinct characteristics of each data structure, (3) minimizes reliance on costly page migrations, and (4) provides memory mapping flexibility beyond the page sizes supported by the system.

4 Chiplet-Locality Aware Page Placement

We present *CLAP*, which provides the suitable page size-specifically, the suitable level of page contiguity-for each data structure.

What is the Suitable Size?: The definition of a suitable page size depends on the system. In this work, we define the suitable page size for MCM GPUs as one that maximizes the benefits of large pages (e.g., reduced address translation overhead) while preserving data locality. As shown in Section 3.3, the best-performing configuration for each workload aligns with this principle. To achieve this, CLAP leverages chiplet-locality, which reflects how many virtually contiguous pages can be grouped together and mapped to the same chiplet. This enables rapid and accurate mapping size selection.

Figure 11 illustrates memory mapping under three different configurations. The upper-left section (①) shows mapping with small pages. Fine-grained mapping enables each data page to be placed at its preferred location, exhibiting strong chiplet-locality (e.g., VPN 0x100–0x103 are mapped to chiplet-0). However, the corresponding physical frames are scattered within the chiplet. The upper-right section (②) shows mapping with large pages. In this case, a contiguous physical frame is reserved for a large page, improving address translation efficiency. However, this coarse-grained allocation forces unrelated regions (e.g., VPN 0x104–0x107) to reside in the same chiplet (chiplet-0), increasing remote memory accesses.

On the other hand, the lower section (③) illustrates CLAP's approach. The system reserves a contiguous physical frame that

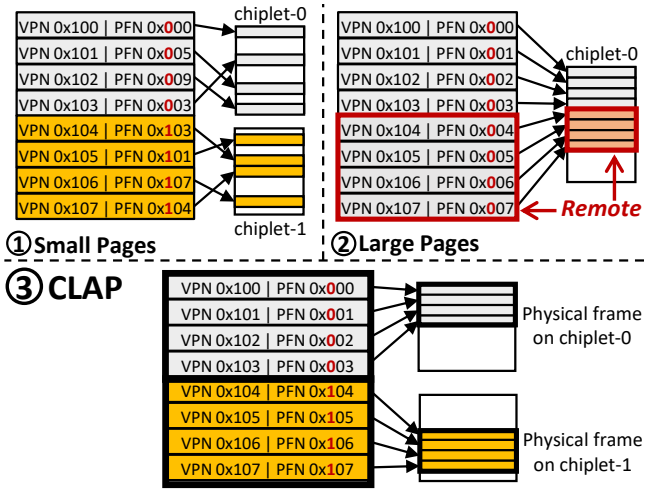


Figure 11: An example of memory mapping without CLAP (top) and with CLAP (bottom). CLAP ensures that chiplet-locality regions are mapped to contiguous physical frames.

matches the size of the chiplet-locality. CLAP then maps virtually contiguous pages to this reserved frame, ensuring alignment between virtual and physical contiguity. This alignment effectively functions as a large page, thereby reducing address translation overhead without compromising memory locality.

Overview: Figure 12 presents an overview of CLAP. *The memory manager* in the GPU driver is responsible for supervising memory mappings for GPU applications. At application launch, the memory manager begins mapping data structures (❶), but limits this to a subset of pages. The process halts once a predefined fraction of the total size has been mapped. This initial phase is referred to as *partial memory mapping* (PMM) (❷).

During PMM, the memory manager uses small pages (e.g., 64KB) and maps each page to the chiplet that requests access. This process creates a sample mapping that serves as the basis for analyzing chiplet-locality. At the same time, the memory manager sends metadata (e.g., allocation ID) to the *Remote Tracker* (RT), a hardware component in the GPU (❸). RT collects runtime metrics, such as the remote access ratio, which are later used to evaluate chiplet-locality and determine the appropriate page size.

Once PMM completes, the GPU driver gathers both the current mapping information and the RT metrics to perform *memory mapping analysis* (MMA) (❹). MMA evaluates the chiplet-locality of

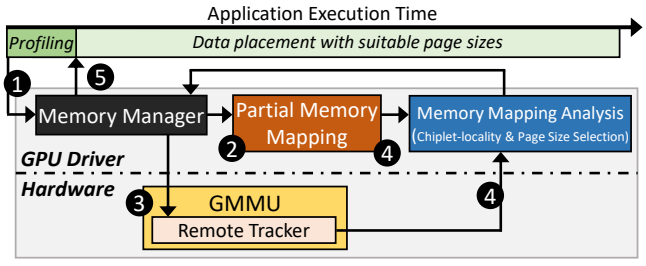


Figure 12: An overview of CLAP.

the data structure and selects the suitable page size. Finally, the memory manager receives the MMA result and applies it to the remaining (unmapped) portion of the data structure (❺).

4.1 Block-based Memory Management

The memory manager employs a *block-based memory management* approach that partitions the virtual address space and physical address space into 2MB units, referred to as **VA (Virtual Address) blocks** and **PF (Physical Frame) blocks**, respectively.

Each VA block serves as a boundary for page size assignment. To support multiple page sizes within a virtual address space, the memory manager assigns a specific page size to each VA block, enforcing that all mappings within the block use that size. For example, assigning a 64KB page size to a 2MB virtual region requires the corresponding VA block to map all of its sub-regions with 64KB pages. This approach simplifies tracking which page size is assigned to each region of the virtual address space.

Similarly to VA blocks, each PF block serves as a boundary for physical frame size assignment. When the memory manager requires a physical frame of a specific size in a particular chiplet, a free PF block within that chiplet is partitioned into multiple frames of the size. For example, with 64KB pages, a free PF block is divided into 32 frames of 64KB, and each frame is inserted into the corresponding free list. The free lists are maintained per chiplet, with each list containing only the frames available in its own chiplet. This policy prevents the intermixing of different frame sizes within a PF block and aligns frames to 2MB boundaries, thereby simplifying frame management and reducing fragmentation.

4.2 Partial Memory Mapping (PMM)

The goal of PMM is to create a sample mapping for chiplet-locality analysis. PMM places data pages that trigger page faults into GPU chiplets until the total number of mapped pages reaches a predefined threshold. During this phase, PMM uses small pages, allowing each data page to be placed in the chiplet that requests access, thereby naturally forming chiplet-locality regions.

In our implementation, PMM uses 64KB pages. This choice is reasonable: 64KB pages provide data locality comparable to 4KB pages while offering higher address translation efficiency (as shown in Figure 6). Additionally, 64KB matches the minimum page migration granularity supported by current commodity GPUs [74].

By default, the PMM threshold is set to 20%, meaning that 20% of each data structure is mapped during PMM, while the remaining 80% is deferred to mapping with the suitable page size. We set the PMM threshold to 20%, a conservative choice empirically derived to ensure reliable chiplet-locality analysis across all evaluated workloads. While a 15% threshold proved sufficient in our experiments, an additional 5% margin was included to enhance robustness. We find that performance is largely insensitive to the PMM threshold. Increasing the threshold to 30% results in only a 1.3% average degradation, suggesting a nonlinear and weak impact. The memory manager in the driver monitors the mapped fraction of each data structure and triggers the memory mapping analysis phase once the threshold is reached.

Opportunistic Large Paging (OLP): Since PMM uses 64KB pages, data mapped in this phase misses the benefits of large pages, even when 2MB mappings would be preferable. To address this, CLAP employs *opportunistic large paging* (OLP), which dynamically forms 2MB pages while preserving fine-grained 64KB mappings during PMM. Figure 13 illustrates the memory mapping flow during PMM with OLP. When the first 64KB page in a VA block is mapped to a chiplet (e.g., chiplet-0), the memory manager reserves a 2MB physical frame for the entire VA block (ⓐ).

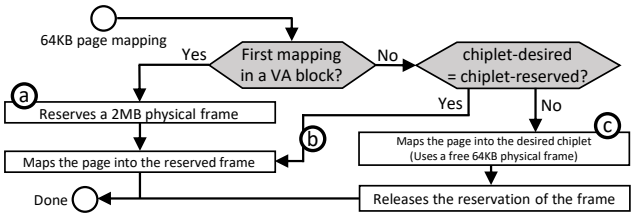


Figure 13: Memory mapping flow during PMM with OLP.

If subsequent 64KB pages from the same VA block are requested by the same chiplet, they are placed into the reserved frame (ⓑ). Once all 64KB pages in the block are mapped to that chiplet, the reserved frame is promoted to a large 2MB page.

However, if a later 64KB page is requested by a different chiplet (e.g., chiplet-1), it is mapped to that chiplet instead (ⓒ). The memory manager then releases the 2MB-frame reservation and places the remaining unused 64KB frames back into the free list, thereby preventing fragmentation and enabling reuse by other VA blocks.

PMM disables OLP and defaults to 64KB mapping if more than 5% of VA blocks release their 2MB reservations. This threshold prevents unnecessary reservation and release operations when opportunistic large paging proves ineffective. The 5% limit is empirically derived to ensure that the 64KB physical frames reclaimed from released reservations (i.e., unused 64KB frames) are fully utilized during subsequent PMM operations.

The duration of the PMM phase varies across data structures depending on their usage patterns—some are used from the beginning of execution, while others are accessed later. We define the PMM phase endpoint as the moment when all data structures have completed profiling. On average, the PMM phase concludes during the early portion of execution, typically within the first 6.92% of kernel instructions executed.

4.3 Remote Tracker (RT)

While PMM establishes a sample mapping for chiplet-locality analysis, RT verifies whether the mapping is truly local to the assigned chiplet. This is because, even with small-page mappings, different cache lines within a page may be accessed by threads on different chiplets, cache lines may be shared across chiplets, or an entire data structure may be globally shared (e.g., matrix B in GEMM).

Figure 14 illustrates the operation of RT. Upon allocation of a GPU data structure, the memory manager assigns an allocation ID. This ID is stored in an unused bit of the last-level page table entry (PTE) [10, 32, 40, 57, 72, 104], allowing RT to identify which pages belong to each allocation. The ID is concurrently forwarded to RT, embedded within the GMMU of each chiplet, and subsequently recorded in its internal table.

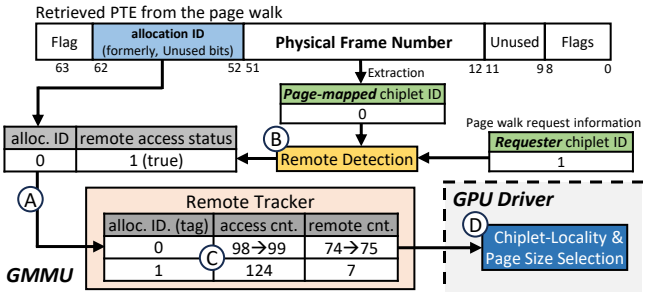


Figure 14: RT monitors remote accesses for each data structure, and the collected statistics are used for chiplet-locality analysis and page size selection.

Once a page walk completes during address translation, an RT table lookup is performed using the allocation ID extracted from the PTE and a remote-access status, which specifies whether the page walk request targets a remote-mapped page (Ⓐ). Since the PFN (Physical Frame Number) in the PTE encodes the chiplet ID to which the page is mapped (i.e., the two MSBs of the channel bits in Figure 4), the request can be classified as local or remote by comparing the page-mapped chiplet ID against the requester chiplet ID from which the memory request originated (Ⓑ). If a match is found in the RT table, RT increments the corresponding access counter and additionally updates the remote counter when the access is identified as remote (Ⓒ). When the PMM phase completes and the driver initiates chiplet-locality analysis for a given allocation (i.e., a data structure), each RT forwards the recorded statistics to the GPU driver and clears the corresponding table entry (Ⓓ). RT communicates with the driver through the host–GMMU interface, which is already employed in GPUs [2, 3].

RT collects statistics based on page walks. Because RT resides within the GMMU—where page walks occur and all required metadata (e.g., allocation ID) is readily available, it avoids the need to propagate this information through packets to other components. RT focuses on estimating the ratio of remote accesses rather than the absolute count or hotness of data structures, making page walk-based counting sufficient for this purpose. We observe that the estimated ratio closely approximates the actual remote access ratio, with a similarity of 95.3%. Each RT includes a 32-entry table in our baseline. This configuration is derived from our evaluation; while a 16-entry table suffices for the evaluated workloads, a 32-entry table provides additional flexibility. When the table is full and a new allocation ID requires insertion, RT replaces the least recently updated entry based on the remote counter. This replacement policy is well aligned with RT's goal of tracking data structures with higher remote access intensity. Upon eviction, the remote ratio of the corresponding entry (i.e., allocation ID) is treated as zero. As an optional feature, evicted entries can be logged to the GPU driver to mitigate data loss, although this option is disabled in our baseline.

Benefits and Overheads: This hardware-based approach offers two advantages over prior software-based profiling techniques [28, 34]. First, RT collects statistics on a per–data structure basis. Second, it eliminates the need for frequent software interventions, which would degrade GPU performance [56]. A single RT requires 288 bytes, with each entry consisting of an 8-bit allocation ID field (baseline) and two 32-bit counters. We estimate the area overhead of a single RT by implementing its Verilog model and synthesizing it with a 28nm UMC standard-cell library [99]. The resulting area is 0.0124mm^2, accounting for approximately 0.0015% of a modern GPU die (~800mm^2) [71, 73]. RT lookup takes only two cycles: one for indexing and one for updating. RT is not on the critical path of memory accesses and does not require inter-RT synchronization.

Modern PTEs include 13 reserved bits (11 unused + 2 ignored) [10, 40, 57, 104]. To demonstrate that these bits suffice to encode allocation IDs, we profile diverse GPU applications—particularly LLM inference[54]—which involve multiple GPU memory allocations. By varying inputs, model types [43, 98, 110], and scaling parameters up to 30 billion, we observe up to around 300 GPU allocations, which is well within the capacity of the reserved bits. This figure reflects the total allocations across the entire execution, so the number of concurrent active allocations is likely much smaller.

4.4 Memory Mapping Analysis (MMA)

MMA infers the chiplet-locality of a data structure from the pages mapped during PMM and determines the suitable page size.

Tree-Based Chiplet-Locality Analysis When the number of mapped data pages for a data structure reaches the PMM threshold (20%), the GPU driver invokes MMA to evaluate chiplet-locality. MMA examines VA blocks of the data structure, in which all 64KB subpages are fully mapped, and computes their chiplet-locality using a tree-based algorithm. Figure 15 illustrates the workflow of the algorithm using a simplified example of a 512KB VA block. The tree structure, which holds page-to-chiplet mapping information, is explicitly maintained in the host and is updated by the memory manager when data pages are mapped to GPUs. The tree's leaf nodes represent contiguous 64KB VA regions, with each node storing the chiplet ID to which its corresponding region is mapped (❶). At each internal node, the chiplet IDs of its child nodes are aggregated into counts (❷). Each internal node is then assigned a locality score ($score(l)$), computed using the following Eq (1), where n denotes the total number of chiplets in the system and l indicates the tree level of the internal node (❸).

$$score(l) = \frac{\max(C_1, C_2, \ldots, C_n)}{\#leaf\_nodes(l)} \quad (1)$$

The locality score at each node quantifies the proportion of its descendant leaf nodes that are mapped to the same chiplet. A score of '1' indicates that all corresponding 64KB pages have been entirely

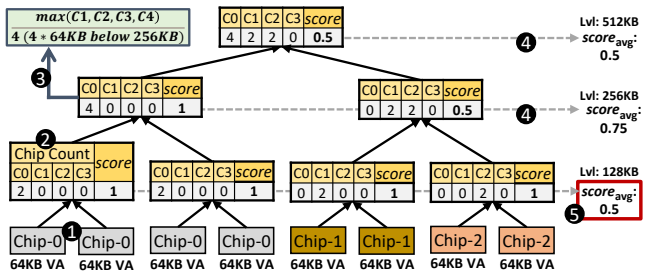


Figure 15: An example of the tree-based algorithm for chiplet-locality analysis on a 512KB VA region.

mapped to a single chiplet. MMA computes the average locality score at each level of the tree ($score_{\text{avg}}(l)$), which represents the proportion of 64KB pages correctly mapped to their target chiplets under a power-of-two page size (❹). Based on these averages, MMA determines the chiplet-locality of the data structure by selecting the highest level in the tree where the average score meets or exceeds a predefined threshold ($thres$), as formulated in Eq (2).

$$\max_{l} \quad l \quad \text{s.t.} \quad score_{\text{avg}}(l) \geq thres \tag{2}$$

By default, this threshold is set to '1', ensuring that all 64KB leaf pages beneath the selected level (i.e., the selected size) are mapped entirely to their corresponding chiplets (❺). This allows MMA to identify the largest page size that preserves chiplet-locality. When analyzing multiple VA blocks, MMA computes the chiplet-locality of each VA block and selects the most dominant degree as the chiplet-locality for the data structure.

With Remote Tracker: In some cases, a data structure may inherently incur high remote accesses regardless of its mapping strategy due to its shared characteristics between threads (e.g. matrix B in GEMM). In such scenarios, increasing the page size becomes a more effective solution, as it reduces address translation overhead even in the absence of strong locality. To support this, MMA incorporates remote access statistics from RT. It aggregates counter statistics from RTs and computes the remote access ratio ($ratio_{\text{rt}}$) for the data structure. MMA then refines the chiplet-locality selection criterion in Eq (2) by subtracting the remote access ratio $ratio_{\text{rt}}$ from the threshold $thres$. This derivation proceeds as follows.

$$k(thres - score_{\text{avg}}(l)) - ratio_{\text{rt}} \leq ratio_{\text{target}} \tag{3}$$

In Eq.(3), $k(thres - score_{\text{avg}}(l))$ represents the expected remote access ratio when the system uses level l as the page size in memory mapping, where k is a scaling parameter. After subtracting the inherent remote access ratio ($ratio_{\text{rt}}$), the remaining ratio must be no greater than CLAP's target remote access ratio ($ratio_{\text{target}}$). Rearranging Eq (3) yields Eq (4), which replaces the condition in Eq (2). In our implementation, we set $ratio_{\text{target}} = 0$ and $k = 1$.

$$score_{\text{avg}}(l) \geq thres - \frac{ratio_{\text{rt}} + ratio_{\text{target}}}{k} \tag{4}$$

Because the original threshold of '1' enforces that the mapping strategy adheres to chiplet-locality, modifying the selection condition relaxes this constraint, increasing the likelihood of selecting larger page sizes. In Figure 15, if the remote access ratio of the data structure is 0.75, MMA reformulates the selection criterion so that the effective threshold is reduced to 0.25 (i.e., $1.00 - 0.75$), resulting in the selection of the 512KB level as the suitable page size.

Benefits and Overheads: MMA operates in the background within the GPU driver, allowing GPU applications to execute without interruption or stalls. Specifically, the chiplet-locality tree for each 2MB VA block is incrementally updated whenever a leaf node (i.e., a 64KB page) is mapped to GPU memory. We estimate this updating overhead using a CPU implementation, which takes less than 1µs per update (typically around 0.4µs). This latency is entirely overlapped with the page mapping process, including page table updates and GPU memory zeroing, which takes approximately 3µs, as measured using NVIDIA UVM on commodity GPUs [71, 73]. Additional operations, such as locality score calculation and page size selection on the tree structure, incur less than 1µs of overhead and are also fully hidden.

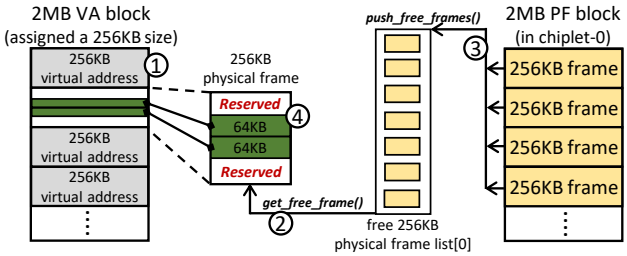


Figure 16: Memory mapping flow with a 256KB page size.

4.5 Applying the Selected Page Size

The memory manager receives the page size selected by MMA and uses it to map the remaining portion of the data structure. The selected page size can span a wide range, including intermediate sizes that are not natively supported by current GPUs (e.g., 128KB to 1MB). In particular, supporting intermediate sizes would require significant changes to both the page table and the GMMU, reducing compatibility with existing GPU systems. To address this, CLAP constructs a large-page-like region by grouping conventional small pages (i.e.g, 64KB pages).

Figure 16 illustrates a memory mapping example with a 256KB page size. A VA block is assigned a 256KB page size, and logically partitioned into multiple 256KB VA regions (①). When the first 64KB page in a 256KB VA region requires mapping to GPU memory (e.g., into chiplet-0), the memory manager retrieves a free 256KB physical frame from the corresponding free list (②). If no free frame is available, the manager fills the list by partitioning a free PF block in that chiplet into multiple 256KB frames (③). The retrieved 256KB frame is reserved for the VA region, and subsequent 64KB sub-regions within that VA region are mapped on demand into the reserved frame using conventional 64KB pages (④). This strategy constructs a 256KB page-like block that is both virtually and physically contiguous.

Handling Edge Cases: In rare cases, CLAP may fail to perform MMA when none of the VA blocks of a data structure are fully mapped during PMM. Similarly, small allocations (typically less than 10MB) may also lack sufficient 2MB VA blocks for meaningful analysis. In such scenarios, CLAP falls back to *opportunistic large paging* (OLP), as described in Section 4.2. OLP dynamically exploits chiplet-locality by opportunistically forming large pages while maintaining fine-grained mapping, effectively addressing these cases. In our evaluation, using OLP in these edge cases results in only a 1.1% performance difference compared to an ideally selected page size.

4.6 Cooperation with TLB Coalescing

As CLAP supports diverse page sizes (i.e., groups of contiguous pages), it requires corresponding architectural support to fully leverage their benefits. While a 2MB-sized group of contiguous pages can be promoted to a true 2MB page—already supported in current GPUs [69], intermediate-sized groups (e.g., 256KB) cannot. Even if such intermediate sizes were supported, they would require

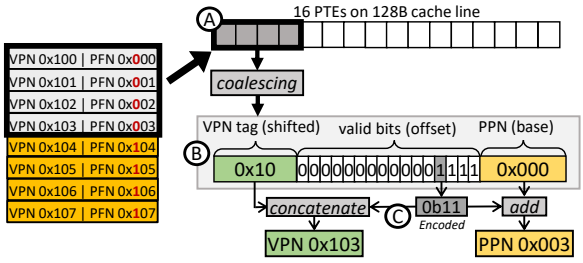


Figure 17: TLB coalescing flow. CLAP leverages coalescing to exploit deliberately mapped contiguous pages, enabling the benefits of larger effective pages.

additional TLB structures, incurring storage overhead and causing resource underutilization when unused. To address this, CLAP adopts a CLAP-specific TLB coalescing mechanism.

Figure 17 illustrates how TLB coalescing collaborates with CLAP, using an example of a 256KB-sized group. When a page walk request for VPN 0x103 is issued, the page walker retrieves the corresponding PTEs and loads them into the L2 cache. Because GPUs use 128B cache lines (consisting of four 32B sectors), sixteen 8-byte PTEs are fetched and packed into a single cache line (Ⓐ). The coalescing logic in the TLB controller then inspects the fetched PTEs to detect contiguous mappings. Once contiguity is confirmed, these PTEs are coalesced and inserted into the TLB as a single coalesced entry (Ⓑ). This entry records the covered virtual range using a set of valid bits, and a coalesced TLB entry can represent up to sixteen contiguous pages (up to 1MB). In case of a TLB hit in a coalesced entry, the TLB controller reconstructs the corresponding VPN and PPN through bit concatenation and addition (Ⓒ). This approach exploits deliberately mapped contiguous pages, effectively extending the coverage of a single TLB entry. In the case of a 2MB-sized group, the memory manager promotes it to a 2MB page [69] and inserts it into the TLB for 2MB pages.

Benefits and Overheads: Since CLAP employs reservation-based mapping, pages within a reserved region preserve the same virtual-to-physical offset. This allows the hardware to coalesce even partially contiguous PTEs into a single entry. To estimate the hardware cost of supporting CLAP's coalescing mechanism, we evaluate two components. First, we implement Verilog models of the additional TLB controller logic (i.e., coalescing, concatenation, and addition units) and synthesize them using the 28nm UMC standard-cell library [99]. The synthesized design occupies 0.0024mm^2, accounting for 0.0003% of the GPU die area [71, 73]. Second, we estimate the extended L2 TLB memory-cell area using CACTI [19], finding that the modification increases the area by 8.98%. For comparison, increasing the L2 TLB size by an equivalent area overhead yields only a 2.31% performance improvement, demonstrating that CLAP achieves higher efficiency.

4.7 Discussion

Memory Fragmentation: CLAP is designed to mitigate both external and internal fragmentation through the following design choices. Because CLAP determines the suitable page size for each data structure, the memory manager maps that structure exclusively with physical frames of the chosen size. Accordingly, the manager can maintain a dedicated free list per data structure. When a free PF block is split into multiple free frames, all frames are inserted into the same list, ensuring that the PF block is used by a single data structure. As a result, when the structure is deallocated, the manager can reclaim the entire PF block and reassign it to other data structures without fragmentation.

Internal fragmentation may occur when a reserved physical frame is not fully utilized. However, CLAP reserves physical frames exclusively for virtual address regions that are at least partially mapped to GPU memory. Such regions are substantially more likely to be fully utilized during execution. We evaluate the total number of PF blocks consumed under different paging schemes and find that CLAP increases memory usage by only 0.57% and 1.27% on average, relative to the 64KB and 2MB static paging schemes, respectively.

Scalability: Although our baseline implementation of CLAP adopts 64KB as the base page size, its core mechanisms-including block-based mapping, physical frame reservation, and tree-based chiplet-locality analysis—are scalable to both smaller and larger base page sizes. For example, CLAP can be adapted to support 4KB base pages, enabling finer-grained page sizes between 4KB and 64KB. Conversely, by increasing the base page size to 2MB and adjusting the VA and PF block sizes to 1GB, CLAP can support large pages up to 1GB, maximizing the benefits of large-page mappings.

Data Initialization: For a data structure, memory access patterns may differ between the initialization and computation phases. However, GPU workloads typically initialize input data on the CPU.Output data structures are inherently part of the computation phase and thus naturally follow the kernel's memory access pattern. These features allow CLAP to capture representative memory access patterns during profiling.

Chiplet Memory Exhaustion: A potential corner case arise when a group of pages needs to be mapped into a chiplet whose physical frames are fully occupied. In such cases, if free frames exist on another chiplet, CLAP maps the pages to those frames instead. While this reduces memory locality, migrating already-mapped pages would both degrade locality and incur additional overhead, such as TLB shootdowns. This scenario is rare in practice because GPU data structures are typically accessed in a balanced manner by threads across chiplets. As a result, when mappings follow locality-as in CLAP-they naturally lead to well-distributed pages across chiplets, consistent with previous observations [108].

Another case can arise when (1) all chiplets' memory spaces are fully utilized, such as under memory oversubscription in UVM [3, 11, 17, 33, 50, 62, 91], or (2) pages of a data structure must be mapped to a specific chiplet because the structure is in the profiling phase. In the first case, CLAP migrates page groups, whose size matches that of the group currently being mapped, to the host memory. The memory manager selects migration targets from pages whose profiling has already completed, prioritizing those least recently mapped to the GPU [3]. In the second case, CLAP employs the same migration mechanism, but pages can instead be migrated to another chiplet if free physical frames are available there. The destination chiplet is selected as the one with the fewest mapped pages, thereby improving balance in memory usage across chiplets.

Cooperation with a Multi-Page TLB: While our baseline assumes separate TLBs for each page size (e.g., a 64KB TLB and a 2MB TLB), CLAP can also operates with multi-page TLB designs [26,

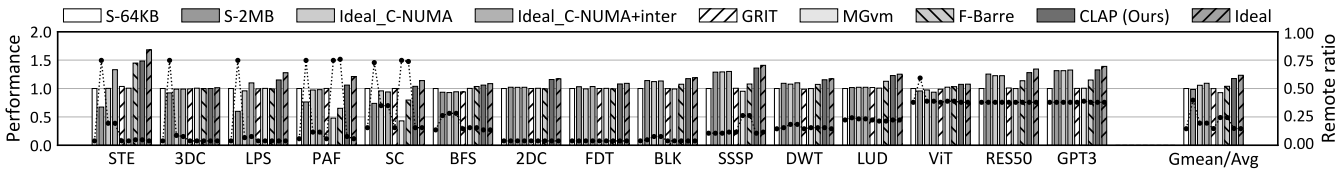


Figure 18: Performance (bar, normalized to S-64KB) and remote access ratio of memory instructions (line) for different configurations.

84, 92], which store TLB entries of different page sizes in a single structure. Since prior work have suggested how the TLB coalescing mechanism [86] can be applied to multi-page TLBs [26], CLAP can naturally build upon these techniques to extend its own coalescing mechanism to such designs.

5 Evaluation

We compare CLAP with eight alternative configurations, as detailed below. Unless otherwise specified, all configurations use 64KB as the base page size. For prior works, we adopt the strategies proposed in each paper to determine the target chiplet for each data page, while preserving demand paging.

1 & 2. Static Paging (S-64KB & S-2MB): These configurations employ a fixed page size (either 64KB or 2MB) for memory mapping. All other system settings remain consistent with the baseline.

3 & 4. Ideal C-NUMA (Ideal_C-NUMA & Ideal_C-NUMA+inter): C-NUMA [28, 34] dynamically adjusts page sizes during runtime through page migration. In our evaluation, we implement an idealized version of C-NUMA (*Ideal_C-NUMA*), by assuming zero latency for all migration-related operations, including TLB shootdowns and cache flushes. We also evaluate a hypothetical variant that supports intermediate page sizes by gradually adapting them based on runtime behavior (*Ideal_C-NUMA+inter*).

5. GRIT: GRIT [104], originally proposed for multi-GPU systems, records page access history and leverages it to guide page migration for improved data locality. We adapt GRIT for MCM GPU systems, omitting page duplication, which is not feasible under a unified page table architecture. As with Ideal C-NUMA, we assume ideal page migration with zero latency.

6. MGvm: MGvm [87] focuses on enhancing address translation efficiency in MCM GPUs. It reduces remote accesses along the address translation path by optimizing the placement of PTE pages and the distribution of TLB entries across chiplets.

7. Barre-Chord (F-Barre): Barre-Chord [32] reduces address translation overhead in MCM GPUs by coalescing translations for uniformly interleaved pages across multiple chiplets.

8. CLAP (Ours): CLAP determines the suitable page size for each data structure by leveraging chiplet-locality. To support intermediate page sizes, we enhance the 64KB TLBs with coalescing.

9. Ideal: We define an 'Ideal', where data pages are mapped using small sizes (i.e., 64KB) but provides address translation as large pages (i.e., 2MB). This setup enables fine-grained data placement while maintaining high translation efficiency.

5.1 Performance

Figure 18 presents the overall performance and remote access ratio across all configurations. CLAP outperforms static paging schemes, achieving an average performance gain of 17.5% over 64KB pages and 19.2% over 2MB pages.

Detailed Analysis: Table 4 shows the page sizes selected by CLAP for the three largest data structures in each workload. These structures consistently correspond to those most frequently accessed. The selected page sizes generally align with the trends observed in Figure 6. Supporting a suitable page size (i.e., the suitable level of page contiguity) for each workload helps maintain the remote access ratio at its lowest level. For instance, in the case of BFS, the static 2MB paging scheme increases the remote access ratio due to suboptimal mapping of the third-largest data structure. CLAP mitigates this issue by assigning a 64KB page size to that structure. A similar pattern is observed in ViT.

Cases where opportunistic large paging (OLP) is applied due to insufficient chiplet-locality analysis (i.e., MMA failure) are indicated in italic and bold text in Table 4. Such cases typically occur in small data structures (e.g., less than 10MB), where limited memory size prevents reliable analysis. CLAP effectively handles these structures using OLP, which uses 64KB pages for fine-grained mapping, as observed in PAF, SC, and BFS. In contrast, LUD contains large data structures but maps only a portion of multiple 2MB VA blocks during PMM, resulting in insufficient analysis. In this case, CLAP applies OLP to dynamically construct large pages, ultimately providing 2MB pages. These results demonstrate that CLAP can effectively provide appropriate paging even in edge cases.

In ML workloads, matrix-B, shared across all threads in a kernel, is assigned 2MB large pages by CLAP, which is beneficial for inter-chiplet shared data structures. In contrast, matrix-A and matrix-C result in insufficient chiplet-locality analysis, as only partial regions of multiple 2MB VA blocks are mapped during PMM. However, CLAP's dynamic chiplet-locality tracking via OLP successfully assigns appropriate page sizes for these matrices. For GPT3, which includes a large matrix-A (2.3GB) and a moderate matrix-B (96MB), OLP results in 2MB pages for both. In contrast, for ViT, which has a small matrix-A (3MB) and a moderate matrix-C (32MB), OLP provides 64KB and 2MB page sizes, respectively.

Compared to NUMA Paging in CPUs: CLAP outperforms ideal C-NUMA and its variant with intermediate page size support by

Abbr.	Page Size	Abbr.	Page Size	Abbr.	Page Size
STE	256KB/256KB	3DC	64KB/64KB	LPS	256KB/256KB
PAF	128KB/***64KB***/***64KB***	SC	128KB/***64KB***/***64KB***	BFS	2MB/2MB/***64KB***
2DC	2MB, 2MB	FDT	2MB, 2MB, 2MB	BLK	2MB, 2MB, 2MB
SSSP	2MB, 2MB, 2MB	DWT	2MB, 2MB, 2MB	LUD	***2MB***
ViT	matrix A/B/C ***64KB***/2MB/ ***2MB***	RES50	matrix A/B/C ***2MB***/2MB/ ***2MB***	GPT3	matrix A/B/C ***2MB***/2MB/ ***2MB***

Table 4: The determined page sizes with CLAP for the three largest data structures in each workload. Italic and bold text indicates paging results with OLP.

11.9% and 8.5%, respectively. C-NUMA adjusts page sizes globally, without considering individual data structures, which is suboptimal when different structures within a workload prefer different sizes, as observed in BFS and ViT. Moreover, its lack of support for intermediate page sizes further limits performance, especially in STE, LPS, PAF, and SC. While intermediate page size support for C-NUMA helps narrow the gap, as seen in STE and LPS, a performance disparity remains. This is because C-NUMA cannot quickly identify the appropriate page size, instead requiring additional time to converge. In contrast, CLAP efficiently assigns appropriate page sizes by leveraging chiplet-locality. Additionally, C-NUMA's optimization fundamentally relies on page migration within or between chiplets. These migrations trigger frequent TLB shootdowns and cache flushes, diminishing overall memory system efficiency, even without explicitly modeling their latency.

Compared to Prior Work in Multi-GPUs: CLAP outperforms GRIT by 17.1%. While GRIT maintains high data locality by placing pages in appropriate chiplets, its performance gains are limited due to its fixed page size. Without page size adaptation, GRIT cannot exploit the benefits of large pages, resulting in performance nearly identical to the static 64KB paging scheme. In contrast, CLAP combines high data locality with large page support.

Compared to Prior Works in MCM GPUs: CLAP outperforms MGvm by 24.8% and F-Barre by 13.8%. MGvm enhances TLB efficiency by optimizing the placement of TLB entries, whereas CLAP achieves greater efficiency by leveraging larger page sizes. F-Barre identifies memory mapping patterns across interleaved pages distributed among chiplets and utilizes them to improve translation efficiency. In contrast, CLAP provides more consistent benefits by coalescing deliberately mapped contiguous pages or promoting them into large pages, which is particularly effective when considering data locality across adjacent pages [13, 37, 103]. Importantly, while both MGvm and F-Barre support multiple page sizes, they lack flexibility: neither do not support intermediate page sizes and both apply a single, global page size across all data structures without page size adaptation.

Compared to Ideal: The average performance gap between CLAP and Ideal is 5.78%. Workloads such as 2DC, FDT, and BLK approach the Ideal because they inherently benefit from large pages. In contrast, workloads such as STE, LPS, and SC exhibit relatively larger gaps, as they favor smaller page sizes to preserve data locality. Because small pages cannot provide the same level of address translation efficiency as 2MB pages, this gap remains, even though CLAP narrows it by supporting intermediate page sizes. This gap cannot be closed by page size selection alone, but it could be further reduced through complementary techniques such as address translation optimizations [46, 55, 59, 85, 93, 94].

5.2 Sensitive Analysis

Applying to Static-Analysis-based Approach: While CLAP utilizes a profiling-based method with first-touch–based page mapping (FT policy) [13], its contribution is not tied to the mapping strategy itself; it can be extended to work with other strategies, such as static-analysis–based approaches (SA policy) [17, 47].

To demonstrate this, we first implement SA policy by combining two prior techniques: LASP [47] and SUV [17]. LASP identifies

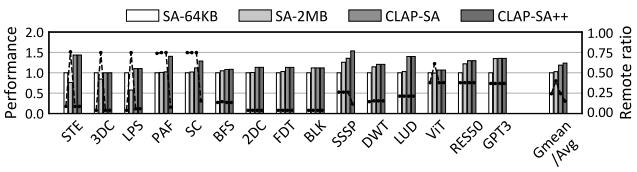


Figure 19: Performance (bar, normalized to SA-64KB) and remote access ratio (line) for configurations based on static approaches [17, 47]. CLAP-SA maximizes the benefit of static approaches through suitable page size selection, while CLAP-SA++ extends support for irregular workloads through runtime profiling.

locality patterns in the relationship between threadblocks and data using code-level variables, whereas SUV leverages LLVM IR [63] to statically compute the memory ranges accessed by each thread. Based on these analyses, SA policy places threadblocks and data pages according to their predicted mappings. We then replace the memory mapping policy in CLAP's profiling phase with SA policy, while retaining the same tree-based algorithm for MMA **(CLAP-SA)**. CLAP-SA selects the appropriate page size based on the statically predicted placement and sharing patterns of data pages, and applies this page size for memory mapping.

Determining the suitable page size remains critical under SA policy. As shown in Figure 19, which compares CLAP-SA against SA policies using fixed 64KB and 2MB page sizes **(SA-64KB & SA-2MB)**, workloads such as STE, 3DC, and LPS exhibit high remote accesses when using 2MB pages. This occurs because, even if the placement range of a data structure is statically identified, the actual mapping must conform to page granularity; thus, an inappropriate page size can cause unintended misalignment. CLAP-SA addresses this by selecting a suitable page size that preserves the predicted data placement while adding flexibility through support for intermediate sizes. With these benefits, CLAP-SA achieves average speedups of 18.8% and 16.1% over SA-64KB and SA-2MB, respectively. This result demonstrates the effectiveness of CLAP's core idea-providing the suitable page size-even under a different approach. Notably, it also exhibits the potential applicability of CLAP to diverse programming models, including conventional copy-and-execute models without demand paging, where static profiling can be applied proactively prior to kernel launch.

In addition, a static-only approach has a fundamental limitation: it cannot effectively analyze data structures with irregular memory access patterns, such as those involving pointer chasing. This leads to high remote access ratios for workloads such as PAF, SC, and SSSP. To overcome this, we enhance CLAP-SA by incorporating CLAP's runtime profiling for data structures with irregular patterns **(CLAP-SA++)**. CLAP-SA++ complements the static-only approach by handling irregular CLAP-SA++ complements the static-only approach by handling irregular workloads, achieving average speedups of 23.7% and 21.0% over SA-64KB and SA-2MB, respectively, and reducing the remote access ratio to 13.6Overall, this presents a hybrid solution that combines static analysis with brief runtime profiling.

Cooperation with Page Migrations: Dynamic adjustment through page migration can be effective when the usage pattern of a data

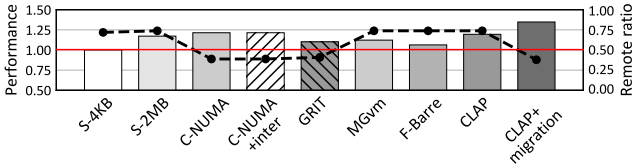


Figure 20: Performance (bar) and remote access ratio (line) for different configurations. The workload (GEMM, 8192 × 768 × 1024) reuses a portion of the data structure from previous kernels, resulting in a different memory access pattern.

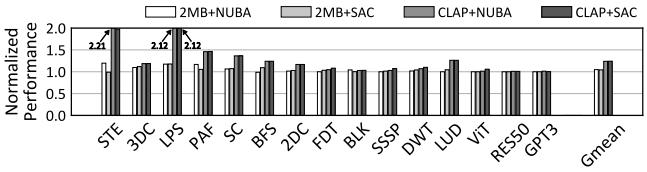


Figure 21: Performance of various remote caching schemes under static 2MB paging and CLAP. Results are normalized to static 2MB paging without remote caching.

structure changes during execution, particularly when the structure is shared across multiple GPU kernels. In practice, however, such dynamic changes are relatively uncommon, as memory access patterns tend to remain consistent within an application [47]. Nevertheless, to explore CLAP's potential in rare but impactful scenarios, we investigate an extension that incorporates page migration. We consider *a challenging scenario for CLAP*, where a data structure is reused across kernels with a different memory access pattern. As a concrete example, we assume that the output matrix $\mathbf{C}^*$ from a GEMM operation is reused in the subsequent GEMM kernel, but only one quarter of the matrix is accessed.

Figure 20 presents a performance comparison of different configurations, explicitly accounting for page migration overhead [45]. CLAP improves performance by 19.4% by selecting appropriate page sizes for newly mapped data structures. However, it cannot remap matrix $\mathbf{C}^*$ assigned in the previous kernel, resulting in high remote accesses. In contrast, C-NUMA and GRIT mitigate remote accesses by relocating matrix $\mathbf{C}^*$, with a few page migrations, achieving speedups of 22.4% and 10.1%, respectively. In this context, we extend CLAP with C-NUMA-based page migration (**CLAP+migration**), applying migration only to data structures shared across multiple kernels. This selective approach successfully remaps matrix $\mathbf{C}^*$, reduces remote accesses, and improves performance by 34.7%. Because CLAP already avoids unnecessary migrations through appropriate page size selection, combining it with selective migration provides a practical alternative for dynamic workloads.

Cooperation with Prior Caching Schemes: CLAP is orthogonal to various prior techniques [64, 90, 106, 109, 111], enabling further improvements in their efficiency. We integrate two state-of-the-art remote caching schemes (NUBA [111] and SAC [109]) and compare their performance under static 2MB paging and CLAP. As shown in Figure 21, CLAP significantly boosts their effectiveness by reducing remote accesses prior to caching (NUBA: 4.8% to 23.9%, SAC: 4.3% to 24.1%), allowing each scheme to better realize its potential.

Number of Chiplets: We evaluate CLAP on an 8-chip MCM GPU, excluding 3DC and SC due to insufficient parallelism (i.e., too few

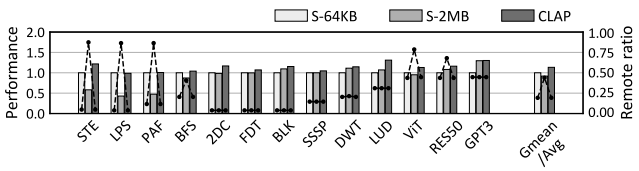


Figure 22: Performance (bar, normalized to S-64KB) and remote access ratio (line) comparison in an 8-chip MCM design.

TBs per kernel to fully utilize all chiplets). As shown in Figure 22, CLAP achieves 13.3% and 21.5% speedups over static 64KB and 2MB paging schemes. Notably, the performance gap between CLAP and the 2MB paging scheme widens compared to the 4-chiplet configuration, indicating that indiscriminate use of large pages becomes detrimental as system scale grows. These results underscore the importance of adaptive page size management and reinforce CLAP's relevance for future large-scale MCM GPUs.

6 Related Work

Optimizing Multi-GPU Systems: Numerous studies have proposed strategies to mitigate memory non-uniformity in multi-GPU systems through optimized thread and data placement [49, 102], improvements in inter-GPU communication [30, 65, 66], remote caching mechanisms [64, 106], and page migration strategies [20, 104, 105]. We expect that our findings can complement and enhance the effectiveness of these approaches, even in different types of NUMA GPU systems.

Optimizing Address Translation in GPUs: Prior work has focused on reducing address translation overhead in GPUs by extending TLB reach [15, 16, 41, 52, 56, 58, 59, 61], optimizing page walks [31, 42, 55, 57, 60, 93, 94], and applying speculative translation [85]. These techniques are orthogonal to CLAP and can be integrated to further reduce address translation costs.

7 Conclusion

We show that page size selection in memory mapping plays a critical role in MCM GPUs, as inappropriate sizes can exacerbate data misplacement across chiplets and amplify memory non-uniformity. To address this challenge, we propose CLAP, a mechanism that optimizes page sizes to harness the benefits of large pages without compromising locality. CLAP leverages chiplet-locality, a unique property of GPU data structures, to enable accurate and efficient page size selection. Our evaluation shows that CLAP improves performance by up to 19.2% over prior paging schemes.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback that helped improve the quality of our paper. This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.10692981, 25% / No.00228970, 50%), Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government(MSIT) [25ZS1100, Research on High-Performance Computing to overcome Limitations of AI], and the BK21 FOUR Project. Sungbin Jang, Osang Kwon, Yongho Lee, and Seokin Hong are affiliated with the Department of Electrical and Computer Engineering. Seokin Hong is the corresponding author.

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 283–300. https://doi.org/10.1145/3373376.3378468

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

[3] Tyler Allen and Rong Ge. 2021. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Article 64, 15 pages. https://doi.org/10.1145/3458817.3480855

[4] AMD. 2021. AMD CNA2 ARCHITECTURE. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.

[5] AMD. 2023. 4th Gen AMD EPYC Processor Architecture White Paper. [Online]. Available. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/221704010-B_en_4th-Gen-AMD-EPYC-Processor-Architecture---White-Paper_pdf.pdf

[6] AMD. 2023. AMD CNA3 ARCHITECTURE. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf.

[7] AMD. 2023. AMD EPYC 9004 Series Processors BIOS and Workload Tuning Guide. [Online]. Available. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58011-epyc-9004-tg-bios-and-workload.pdf

[8] AMD. 2024. 5th Gen AMD EPYC Processor Architecture White Paper. [Online]. Available. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/5th-gen-amd-epyc-processor-architecture-white-paper.pdf

[9] AMD. 2024. AMD EPYC 9005 Series Processors BIOS and Workload Tuning Guide. [Online]. Available. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58467_amd-epyc-9005-tg-bios-and-workload.pdf

[10] AMD. 2024. AMD64 Architecture Programmer's Manual Volumes 1-5. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf.

[11] AMD. 2024. Unified memory management. [Online]. Available: https://rocm.docs.amd.com/projects/HIP/en/docs-develop/how-to/hip_runtime_api/memory_management/unified_memory.html.

[12] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. Article 35, 14 pages. https://doi.org/10.1145/3342195.3387518

[13] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. 320–332. https://doi.org/10.1145/3079856.3080231

[14] Akhil Arunkumar, Evgeny Bolotin, David Nellans, and Carole-Jean Wu. 2019. Understanding the Future of Energy Efficiency in Multi-Module GPUs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 519–532. https://doi.org/10.1109/HPCA.2019.00063

[15] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 136–150. https://doi.org/10.1145/3123939.3123975

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

[17] Pratheek B, Guilherme Cox, Jan Vesely, and Arkaprava Basu. 2024. SUV: Static Analysis Guided Unified Virtual Memory. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 293–308. https://doi.org/10.1109/MICRO61859.2024.00030

[18] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[19] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2, Article 14 (June 2017), 25 pages. https://doi.org/10.1145/3085572

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

[21] Brown, Tom and Mann, Benjamin and Ryder, Nick and Subbiah, Melanie and Kaplan, Jared D and Dhariwal, Prafulla and Neelakantan, Arvind and Shyam, Pranav and Sastry, Girish and Askell, Amanda and Agarwal, Sandhini and Herbert-Voss, Ariel and Krueger, Gretchen and Henighan, Tom and Child, Rewon and Ramesh, Aditya and Ziegler, Daniel and Wu, Jeffrey and Winter, Clemens and Hesse, Chris and Chen, Mark and Sigler, Eric and Litwin, Mateusz and Gray, Scott and Chess, Benjamin and Clark, Jack and Berner, Christopher and McCandlish, Sam and Radford, Alec and Sutskever, Ilya and Amodei, Dario. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. 1877–1901. https://doi.org/10.48550/arXiv.2005.14165

[22] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[23] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R. Johnson, Stephen W. Keckler, Minsoo Rhu, and William J. Dally. 2017. Architecting an Energy-Efficient DRAM System for GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 73–84. https://doi.org/10.1109/HPCA.2017.58

[24] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding Irregular GPGPU Graph Applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195. https://doi.org/10.1109/IISWC.2013.6704684

[25] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

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

[27] Preyesh Dalmia, Rajesh Shashi Kumar, and Matthew D. Sinclair. 2024. CPElide: Efficient Multi-Chiplet GPU Coherence. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[28] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 381–394. https://doi.org/10.1145/2451116.2451157

[29] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv preprint arXiv:2010.11929* (2020).

[30] Amel Fatima, Yang Yang, Yifan Sun, Rachata Ausavarungnirun, and Adwait Jog. 2025. NetCrafter: Tailoring Network Traffic for Non-Uniform Bandwidth Multi-GPU Systems. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*. 1064–1078. https://doi.org/10.1145/3695053.3731040

[31] Yuan Feng, Yuke Li, Jiwon Lee, Won Woo Ro, and Hyeran Jeon. 2025. Heliostat: Harnessing Ray Tracing Accelerators for Page Table Walks. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*. 122–136. https://doi.org/10.1145/3695053.3731011

[32] Yuan Feng, Seonjin Na, Hyesoon Kim, and Hyeran Jeon. 2024. Barre Chord: Efficient Virtual Memory Translation for Multi-Chip-Module GPUs. In *Proceedings of the 51th Annual International Symposium on Computer Architecture (ISCA)*. 834–847. https://doi.org/10.1109/ISCA59077.2024.00065

[33] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 224–235. https://doi.org/10.1145/3307650.3322224

[34] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)*. 231–242. https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud

[35] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a High-Level Language Targeted to GPU Codes. In

2012 Innovative Parallel Computing (InPar). 1–10. https://doi.org/10.1109/InPar.2012.6339595

[36] Faruk Guvenilir and Yale N. Patt. 2020. Tailored Page Sizes. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 900–912. https://doi.org/10.1109/ISCA45697.2020.00078

[37] Dongho Ha, Yunho Oh, and Won Woo Ro. 2023. R2D2: Removing ReDunDancy Utilizing Linearity of Address Generation in GPUs. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/3579371.3589039

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[39] IEEE. 2022. INTERNATIONAL ROADMAP FOR DEVICES AND SYSTEMS. [Online]. Available: https://irds.ieee.org/images/files/pdf/2022/2022IRDS_MM.pdf.

[40] Intel. 2025. Intel 64 and IA-32 Architectures Software Developer's Maunal Volume 3 (3A,3B,3C, and 3D): System Programming Guide. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[41] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–24. https://doi.org/10.1145/3309710

[42] Sungbin Jang, Junhyeok Park, Osang Kwon, Yongho Lee, and Seokin Hong. 2024. Rethinking Page Table Structure for Fast Address Translation in GPUs: A Fixed-Size Hashed Page Table. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 325–337. https://doi.org/10.1145/3656019.3676900

[43] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023). https://doi.org/10.48550/arXiv.2310.06825

[44] Zhixian Jin, Christopher Rocca, Jiho Kim, Hans Kasan, Minsoo Rhu, Ali Bakhoda, Tor M. Aamodt, and John Kim. 2024. Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 885–898. https://doi.org/10.1109/MICRO61859.2024.00070

[45] Konstantinos Kanellopoulos, Rahul Bera, Kosta Stojiljkovic, F Nisa Bostanci, Can Firtina, Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar, Mohammad Sadrosadati, Nandita Vijaykumar, et al. 2023. Utopia: Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1196–1212. https://doi.org/10.1145/3613424.3623789

[46] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. 2023. Victima: Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Resources. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1195. https://doi.org/10.1145/3613424.3614276

[47] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G. Rogers. 2020. Locality-Centric Data and Threadblock Management for Massive GPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1022–1036. https://doi.org/10.1109/MICRO50266.2020.00086

[48] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[49] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. 2018. CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 3 (2018), 1–23. https://doi.org/10.1145/3232521

[50] Junsu Kim, Jaebeom Jeon, Jaeyong Park, Sangun Choi, Minseong Gil, Seokin Hong, Gunjae Koo, Myung Kuk Yoon, and Yunho Oh. 2025. MOST: Memory Oversubscription-Aware Scheduling for Tensor Migration on GPU Unified Storage. *IEEE Computer Architecture Letters (CAL)* 24, 2 (2025), 213–216. https://doi.org/10.1109/LCA.2025.3580264

[51] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 307–319. https://doi.org/10.1145/3079856.3080239

[52] Jagadish B. Kotra, Michael LeBeane, Mahmut T. Kandemir, and Gabriel H. Loh. 2021. Increasing GPU Translation Reach by Leveraging Under-Utilized On-Chip Resources. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1169–1181. https://doi.org/10.1145/3466752.3480105

[53] Osang Kwon, Yongho Lee, Junhyeok Park, Sungbin Jang, Byungchul Tak, and Seokin Hong. 2024. Distributed Page Table: Harnessing Physical Memory as an Unbounded Hashed Page Table. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 36–49. https://doi.org/10.1109/MICRO61859.2024.00013

[54] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)* (Koblenz, Germany). Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[55] Jiwon Lee, Gun Ko, Myung Kuk Yoon, Ipoom Jeong, Yunho Oh, and Won Woo Ro. 2025. Marching Page Walks: Batching and Concurrent Page Table Walks for Enhancing GPU Throughput. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1662–1677. https://doi.org/10.1109/HPCA61900.2025.00123

[56] Jiwon Lee, Ju Min Lee, Yunho Oh, William J. Song, and Won Woo Ro. 2023. SnakeByte: A TLB Design with Adaptive and Recursive Page Merging in GPUs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1195–1207. https://doi.org/10.1109/HPCA56546.2023.10071063

[57] Bingyao Li, Yanan Guo, Yueqi Wang, Aamer Jaleel, Jun Yang, and Xulong Tang. 2023. IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1163–1177. https://doi.org/10.1145/3613424.3614269

[58] Bingyao Li, Yueqi Wang, and Xulong Tang. 2023. Orchestrated Scheduling and Partitioning for Improved Address Translation in GPUs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247943

[59] Bingyao Li, Yueqi Wang, Tianyu Wang, Lieven Eeckhout, Jun Yang, Aamer Jaleel, and Xulong Tang. 2024. STAR: Sub-Entry Sharing-Aware TLB for Multi-Instance GPU. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 309–323. https://doi.org/10.1109/MICRO61859.2024.00031

[60] Bingyao Li, Jieming Yin, Anup Holey, Youtao Zhang, Jun Yang, and Xulong Tang. 2023. Trans-FW: Short Circuiting Page Table Walk in Multi-GPU Systems via Remote Forwarding. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 456–470. https://doi.org/10.1109/HPCA56546.2023.10071054

[61] Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. 2021. Improving Address Translation in Multi-GPUs via Sharing and Spilling Aware TLB Design. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1154–1168. https://doi.org/10.1145/3466752.3480083

[62] Mao Lin, Yuan Feng, Guilherme Cox, and Hyeran Jeon. 2025. Forest: Access-aware GPU UVM Management. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*. 137–152. https://doi.org/10.1145/3695053.3731047

[63] LLVM. 2025. Compiling CUDA with clang. [Online]. Available: https://llvm.org/docs/CompileCudaWithLLVM.html.

[64] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-Aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 123–135. https://doi.org/10.1145/3123939.3124534

[65] Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. 2021. GPS: A Global Publish-Subscribe Model for Multi-GPU Memory Management. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 46–58. https://doi.org/10.1145/3466752.3480088

[66] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. 2023. FinePack: Transparently Improving the Efficiency of Fine-Grained Transfers in Multi-GPU Systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 516–529. https://doi.org/10.1109/HPCA56546.2023.10070949

[67] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2807591.2807626

[68] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2003. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.* 36, SI (2003), 89–104. https://doi.org/10.1145/844128.844138

[69] Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis)Managed: A Novel TLB-Based Covert Channel on GPUs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIACCS)*. 872–885.

[70] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-

whitepaper.pdf.
[71] NVIDIA. 2020. NVIDIA A100. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.
[72] NVIDIA. 2020. *Pascal MMU Format Changes*. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf
[73] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU Architecture. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core.
[74] NVIDIA. 2023. NVIDIA Linux Open GPU Kernel Moudle Source. [Online]. Available: https://github.com/NVIDIA/open-gpu-kernel-modules.
[75] NVIDIA. 2024. CUDA C/C++ SDK Code Samples. [Online]. Available: https://developer.nvidia.com/cuda-code-samples.
[76] NVIDIA. 2024. NVIDIA Blackwell Architecture. [Online]. Available: https://resources.nvidia.com/en-us-blackwell-architecture.
[77] NVIDIA. 2024. NVIDIA Blackwell Platform Arrives to Power a New Era of Computing. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing.
[78] NVIDIA. 2024. NVIDIA Grace Hopper Superchip Architecture Whitepaper. [Online]. Available: https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.
[79] NVIDIA. 2025. NVIDIA NVLink-C2C. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink-c2c/.
[80] Yunho Oh, Keunsoo Kim, Kuk Yoon Myung, Hyun Park Jong, Yongjun Park, Woo Ro Won, and Murali Annavaram. 2016. APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 191–203. https://doi.org/10.1109/ISCA.2016.26
[81] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. 2021. Fast Local Page-Tables for Virtualized NUMA Servers with vMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 194–210. https://doi.org/10.1145/3445814.3446709
[82] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 347–360. https://doi.org/10.1145/3297858.3304064
[83] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 679–692. https://doi.org/10.1145/3173162.3173203
[84] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 210–222. https://doi.org/10.1109/HPCA.2015.7056034
[85] Junhyeok Park, Osang Kwon, Yongho Lee, Seongwook Kim, Gwangeun Byeon, Jihun Yoon, Prashant J. Nair, and Seokin Hong. 2024. A Case for Speculative Address Translation with Rapid Validation for GPUs. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 278–292. https://doi.org/10.1109/MICRO61859.2024.00029
[86] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 258–269. https://doi.org/10.1109/MICRO.2012.32
[87] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2022. Designing Virtual Memory System of MCM GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 404–422. https://doi.org/10.1109/MICRO56248.2022.00036
[88] Hongliang Qu and Zhibin Yu. 2024. WASP: Workload-Aware Self-Replicating Page-Tables for NUMA Servers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1233–1249. https://doi.org/10.1145/3620665.3640369
[89] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in x86 Processors. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1106–1120. https://doi.org/10.1145/3466752.3480062
[90] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 582–595. https://doi.org/10.1109/HPCA47549.2020.00054
[91] Nikolay Sakharnykh. 2017. UNIFIED MEMORY ON PASCAL AND VOLTA. *NVIDIA GTC* (2017).
[92] A. Seznec. 2004. Concurrent Support of Multiple page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers (TC)* 53, 7 (2004), 924–927. https://doi.org/10.1109/TC.2004.21
[93] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 180–192. https://doi.org/10.1109/ISCA.2018.00025
[94] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 352–363. https://doi.org/10.1109/MICRO.2018.00036
[95] Alan Smith, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Samuel Naffziger, Mike Mantor, Mark Fowler Nathan Kalyanasundharam, Vamsi Alla, Nicholas Malaya, Joseph L. Greathouse, Eric Chapman, and Raja Swaminathan. 2024. Realizing the AMD Exascale Heterogeneous Processor Vision : Industry Product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 876–889. https://doi.org/10.1109/ISCA59077.2024.00068
[96] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.
[97] Synopsys. 2024. Synopsys XSR PHY IP. [Online]. Available: https://www.synopsys.com/dw/ipdir.php?ds=dwc_usr-xsr_phy.
[98] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMa: Open and Efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023). https://doi.org/10.48550/arXiv.2302.13971
[99] UMC. 2022. 28 Nanometer. [Online]. Available: https://www.umc.com/upload/media/05_Press_Center/3_Literatures/Process_Technology/28nm_Brochure.pdf.
[100] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091
[101] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, Onur Kayiran, Mitesh Meswani, Indrani Paul, Matthew Poremba, Steven Raasch, Steven K. Reinhardt, Greg Sadowski, and Vilas Sridharan. 2017. Design and Analysis of an APU for Exascale Computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 85–96. https://doi.org/10.1109/HPCA.2017.42
[102] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B. Gibbons, and Onur Mutlu. 2018. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 829–842. https://doi.org/10.1109/ISCA.2018.00074
[103] Kai Wang and Calvin Lin. 2017. Decoupled affine computation for SIMT GPUs. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 295–306. https://doi.org/10.1145/3079856.3080205
[104] Yueqi Wang, Bingyao Li, Aamer Jaleel, Jun Yang, and Xulong Tang. 2024. GRIT: Enhancing Multi-GPU Performance with Fine-Grained Dynamic Page Placement. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1080–1094. https://doi.org/10.1109/HPCA57654.2024.00085
[105] Yueqi Wang, Bingyao Li, Mohamed Tarek Ibn Ziad, Lieven Eeckhout, Jun Yang, Aamer Jaleel, and Xulong Tang. 2025. OASIS: Object-Aware Page Management for Multi-GPU Systems. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1678–1692. https://doi.org/10.1109/HPCA61900.2025.00124
[106] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 339–351. https://doi.org/10.1109/MICRO.2018.00035
[107] Z. Bodek. 2014. Transparent Superpages for FreeBSD on ARM. [Online]. Available: https://www.bsdcan.org/2014/schedule/attachments/281_2014_arm_superpages-paper.pdf.
[108] Shiqing Zhang, Mahmood Naderan-Tahan, Magnus Jahre, and Lieven Eeckhout. 2023. Characterizing Multi-Chip GPU Data Sharing. *ACM Transactions on Architecture and Code Optimization (TACO)* 20, 4 (2023), 1–24. https://doi.org/10.1145/3629521
[109] Shiqing Zhang, Mahmood Naderan-Tahan, Magnus Jahre, and Lieven Eeckhout. 2023. SAC: Sharing-Aware Caching in Multi-Chip GPUs. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. 1–13. https://doi.org/10.1145/3579371.3589078
[110] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open Pre-Trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022). https://doi.org/10.48550/arXiv.2205.01068
[111] Xia Zhao, Magnus Jahre, Yuhua Tang, Guangda Zhang, and Lieven Eeckhout. 2023. NUBA: Non-Uniform Bandwidth GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 544–559. https://doi.org/10.1145/3575693.3575745

[112] Xia Zhao, Guangda Zhang, Lu Wang, Shiqing Zhang, and Huadong Dai. 2025. NearFetch: Saving Inter-Module Bandwidth in Many-Chip-Module GPUs. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1693–1706. https://doi.org/10.1109/HPCA61900.2025.00125

[113] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 345–357. https://doi.org/10.1109/HPCA.2016.7446077