2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00085

GRIT: Enhancing Multi-GPU Performance with Fine-Grained Dynamic Page Placement

Yueqi Wang⋆† , Bingyao Li⋆†, Aamer Jaleel‡, Jun Yang†, Xulong Tang†
University of Pittsburgh†, NVIDIA‡
yuw249@pitt.edu, bil35@pitt.edu, ajaleel@nvidia.com, juy9@pitt.edu, tax6@pitt.edu

***Abstract*—Multi-GPU systems have become popular to cater to the growing demands for high parallelism and large memory capacity. However, the delivered performance is constrained by the non-uniform memory access (NUMA) overhead arising from data sharing and communication across multiple GPUs. Recent multi-GPUs employ unified virtual memory (UVM) to simplify the programming effort. In UVM-enabled multi-GPUs, three popular page placement schemes are adopted to mitigate the NUMA overheads: i) on-touch page migration, ii) access counter-based migration, and iii) page duplication. However, we observe that the preferred page placement scheme varies across i) different applications, ii) different pages of the same application, and iii) even different execution phases of a single page, making it challenging to find a "one-size-fits-all" page placement scheme. To this end, we propose GRIT, which dynamically and automatically determines the appropriate page placement schemes at runtime in a fine-grained manner to enhance multi-GPU performance and scalability. Experimental results indicate that GRIT achieves an average of 60%, 49%, and 29% performance improvements over uniformly adopting on-touch migration, access counter-based migration, and page duplication, respectively.**

I. Introduction

Graphics Processing Units (GPUs) are widely used in modern computing systems to provide accelerated performance for various applications [13], [15]–[17], [20], [26], [37], [46], [47], [49], [50], [53], [57], [64]. Despite the continuous efforts from GPU vendors to increase single GPU parallelism and memory capacity, modern GPUs still struggle to keep up with the rapid growth of dataset sizes and parallelism requirements of applications [51]. To cater to application demands, multi-GPU systems today provide high parallelism and large aggregated memory capacity by connecting multiple GPUs through high-bandwidth connections (e.g., NVLink [21]). For instance, NVIDIA DGX-2 [42] features up to 16 GPUs per node, while AMD equips the TS4 server with four MI25 GPUs [7].

Modern multi-GPU systems generally employ unified virtual memory (UVM) to simplify programming and improve application portability and compatibility [4], [5], [41], [45]. UVM allows GPUs to access data residing in remote physical memory through universal pointers. However, the performance of UVM-enabled multi-GPUs is constrained by the non-uniform memory access (NUMA) overheads arising from data sharing and communication across GPUs. To mitigate NUMA overheads, there are three popular page placement schemes in UVM-enabled multi-GPUs. First, on-touch page migration [44] always migrates pages to the requesting GPU's memory when the pages are not locally present. While this guarantees local page access, significant data-sharing across GPUs can lead to frequent page migrations. Second, counter-based page migration [45] uses an "access counter" to trigger page migration when the access counter reaches a certain threshold (e.g., 256 in NVIDIA Volta GPUs [41], [43]). However, the benefits are often offset by the large number of expensive remote accesses and frequent page table entry invalidations [4], [5], [8], [9]. Third, page duplication replicates pages in the GPU's local memory to facilitate local page reads. However, page duplication redundantly keeps duplicated pages, which can potentially lead to memory oversubscription [22], [24], [38]. Moreover, when a GPU performs a write operation, it has to invalidate all page replicas on other GPUs (called *page write collapse*), incurring additional overheads.

Modern multi-GPU systems uniformly adopt one of the page placement schemes. We plot the performance of uniformly adopting each scheme in Figure 1. We use various benchmarks with representative access patterns and implement all three schemes in MGPUSim [56]. The detailed characteristics of benchmarks are given in Table II, and the simulation configurations are discussed in detail in Section III-B. We also include results called *Ideal*. The *Ideal* is implemented as follows: i) All the page reads, except the first cold page reads, can find the page in the GPU's local memory. ii) All the page writes update the pages with zero NUMA latency regardless of whether the pages are in local memory or remote memory. Note that, the *Ideal* is not practical, and we only use it to reflect the optimization potentials. The results in Figure 1 are normalized to the on-touch migration. The figure reveals that there is no "one-size-fits-all" page placement scheme that universally yields the highest performance across all applications. These performance differences are attributed to the diverse page-sharing patterns observed both across different applications and across different execution phases in the same application (quantitative results and detailed analysis are given in Section IV-B). This figure clearly demonstrates the importance of developing a dynamic page placement scheme in multi-GPU systems.

Prior works mitigating NUMA overheads in multi-GPUs include prefetching [23], [24], [52], dynamic page migration [10], [12], [24], and peer-to-peer load/store [38], [39]. Specifically, prefetching relies on the accurate prediction of data access patterns which are hard to observe in GPUs due to the massive parallelism. Dynamic page migration proactively

⋆ The authors contributed equally.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

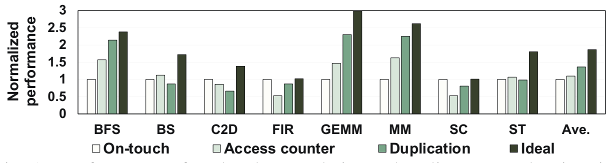


Fig. 1. Performance of each scheme relative to baseline on-touch migration.

migrates the pages in a bandwidth-aware fashion. However, it is solely based on one migration policy (e.g., on-touch), which is not able to benefit different sharing patterns across GPUs. Peer-to-peer load/store allows fine-granular data transfer among GPUs. However, it increases the access latency and requires maintaining expensive cache coherence across multiple GPUs [38], [39].

In this paper, we aim to enhance multi-GPU memory access efficiency by dynamically determining page placement schemes in a fine-grained manner. To this end, We propose fine-**GR**ained dynam**I**c page placemen**T** (GRIT) with three key and novel designs. First, we introduce a Fault-Aware Initiator that detects and decides when to change page placement schemes by leveraging the number of page faults sent to the host. Second, we propose a software-based Page Attribute Table (PA-Table) in memory that records the page access information to determine appropriate page placement schemes. We design a hardware-based Page Attribute Cache (PA-Cache) to reduce the potential memory bandwidth contention caused by memory accesses to PA-Table. Finally, we develop a Neighboring-Aware Predictor that leverages the access pattern similarity of neighboring pages to proactively predict and determine page placement schemes for adjacent pages. The paper makes the following major contributions:

- We conduct a comprehensive characterization and root the source of performance variation across different page placement schemes. We quantitatively analyze the page sharing patterns across different applications and across different execution phases within a single application. The results indicate that i) different pages prefer different placement schemes during execution and ii) neighboring pages generally show the same scheme preferences.
- We propose GRIT, which incorporates three optimizations: i) Fault-Aware Initiator to detect the inappropriate page placement scheme and initial scheme change when necessary; ii) Page Attribute Table to track the page access characteristics and make decisions on the appropriate page placement scheme; and iii) Neighboring-Aware Prediction to proactively determine the page placement scheme for adjacent pages.
- We evaluate GRIT using eight multi-GPU applications. Experimental results show that GRIT achieves an average of 60%, 49%, and 29% performance improvement over on-touch migration, access counter-based migration, and page duplication, respectively. We also evaluate GRIT with large page size and different numbers of GPUs, and compare it with the state-of-the-art. The results show that GRIT either outperforms existing approaches or can be combined with existing approaches to yield further benefits.

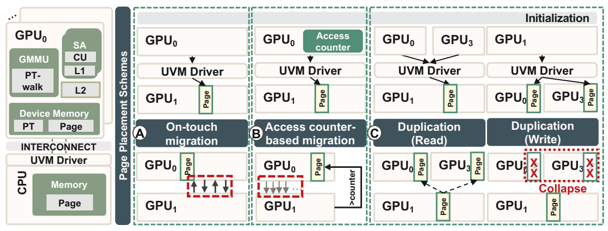


Fig. 2. Baseline GPU architecture and page placement schemes.

II. Background

A. Baseline UVM-Enabled Multi-GPU System

In this paper, we focus on multi-GPU systems, where multiple GPUs are connected via high-bandwidth interconnects such as PCIe [40] or NVLink [21]. Figure 2 shows the multi-GPU architecture. Unified virtual memory (UVM) [21] is employed and managed by the UVM driver on the CPU side. UVM allows programs to use universal pointers to access the memory in the CPU and different GPUs. Each GPU has its own local memory and local page table (PT). If the page table entry is invalid in the local page table, a local page fault is generated and sent to the UVM driver. The UVM driver maintains a centralized page table that stores all valid and up-to-date address translations for all GPUs [5].

B. Page Placement Scheme

There are three common page placement schemes which are illustrated in Figure 2.

1) On-touch migration: As shown in Figure 2 Ⓐ, whenever a GPU accesses a page that is not currently presented in its local memory, the page will be migrated into the requesting GPU memory. Migrating a page is expensive and introduces execution overheads [10], [30], [32], [60]. The detailed page migration is similar to previous works [10], [30], [32], [60]. Specifically, first, UVM flushes in-flight instructions in the CU pipeline and the contents of the caches and TLBs of the GPU that owns the page. Second, the UVM driver migrates the page to the requesting GPU and notifies the finish of the page migration to the requesting GPU. On-touch page placement ensures that the subsequent GPU accesses to the same page can fetch the data from its local memory. However, if a page is accessed by different GPUs in an interleaved manner, on-touch placement can lead to "ping-pong" page migrations, which significantly degrades performance.

2) Access counter-based migration: NVIDIA Volta GPUs and newer generations [14], [41] feature hardware-implemented access counters to avoid frequent page migration. The access counters track the number of remote accesses at a page group granularity of 64 KB. The GPU establishes the address translation to a remote physical page in its local page table. A static threshold of 256 remote accesses is employed to trigger page migration. Figure 2 Ⓑ shows the process of the access counter-based page placement scheme. The detailed process is as follows. 1) When a GPU first accesses a physical page in another GPU, it generates a local page fault. This local page fault is forwarded to the UVM driver. The UVM driver walks the centralized page table to obtain

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

the translation of the page and sends the translation back to the requesting GPU; 2) Upon receiving the translation, the GPU updates the local page table to establish the remote translation mapping, as well as fetches the data at a cache line granularity from the remote GPU; 3) On each remote memory access to a page, the access counter of the corresponding page group is incremented. When the access counter reaches the threshold, the page migration request is generated to the UVM driver; 4) The UVM driver broadcasts the invalidation requests to every GPU to invalidate the page table entries, TLBs, and caches to ensure translation and data coherence, as well as flush in-flight instructions in the CU pipeline; 5) After all invalidations finish, the UVM driver initiates the page migration. This scheme helps reduce "Ping-Pong" page migrations across GPUs but introduces remote access and invalidation overheads [30].

3) Page duplication: Figure 2 Ⓒ shows the page duplication scheme. Specifically, when a read/load operation generates a local page fault, the GPU replicates the page in its physical memory. In scenarios where the page is loaded by multiple GPUs, each GPU stores a page replica in its local memory. When a GPU performs a write operation on a shared page, page write-collapse is required to ensure consistency. Specifically, when any GPU writes a page, the GPU sends a *page protection fault* to the UVM. Upon receiving the page protection fault, the UVM driver sends the page invalidation requests to the corresponding GPUs. GPUs that own the page flush in-flight instructions in the CU pipeline, the contents of TLBs and caches, and invalidate the corresponding Page Table Entry (PTE). Then, the requesting GPU can resume its write operations on the page. If any other GPUs read that page again, the UVM driver copies the most recent version of the page to that particular GPU. The page duplication allows read-shared pages to be accessed locally by multiple GPUs, avoiding remote memory access latency. However, the overhead of collapsing read-write shared pages can be expensive, making it unsuitable for write-intensive applications. Moreover, it is subject to memory oversubscription because of duplicated copies of the same page (as shown by prior works [38], [63]).

III. Methodology

A. Applications

We use eight applications with various multi-GPU memory access and page sharing patterns from AMDAPPSDK [6], Hetero-Mark [55], SHOC [18], and DNN-MARK [20] benchmark suites as listed in Table II. Specifically, `BFS` and `BS` demonstrate a random access pattern where each GPU performs read and write operations to other GPUs in an unpredictable manner. `C2D`, `FIR`, `SC`, and `ST` exhibit adjacent access pattern in which the input data is batched and shared with the neighboring GPUs. `GEMM` and `MM` follow a scatter-gather access pattern where each GPU reads or writes data from local and remote GPUs.

TABLE I
Baseline multi-GPU configuration.

Module	Configuration
Compute Unit	1.0 GHz, 64 per GPU
L1 Vector Cache	16 KB, 4-way
L1 Inst Cache	32 KB, 4-way
L1 Scalar Cache	16 KB, 4-way
L2 Cache	256 KB, 16-way
DRAM	Configured to 70% of application's memory footprint [10], [22], [24], [34]
L1 TLB	32 entries, 32-way, 1-cycle lookup latency, CU private, LRU replacement policy
L2 TLB	512 entries, 16-way, 10-cycle lookup latency, CUs shared, LRU replacement policy
Page table walk	GMMU 8 shared page table walker [48], [54], [59], 100-cycle latency per level [25]
Page walk cache	128 entries shared across page table walker [48]
Page walk queue	64 entries
Access counter threshold	256 [43]
Inter-GPU network	300GB/s NVLink-v2
CPU-GPU network	32GB/s PCIe-v4

TABLE II
List of applications.

Abbr.	Application	Benchmark Suite	Access Pattern	Memory Footprint
`BFS`	Breadth-first Search	SHOC	Random	32 MB
`BS`	Bitonic Sort	AMDAPPSDK	Random	30 MB
`C2D`	Convolution 2D	DNN-Mark	Adjacent	94 MB
`FIR`	Finite Impulse Resp.	Hetero-Mark	Adjacent	155 MB
`GEMM`	General Matrix Multiplication	AMDAPPSDK	Scatter-Gather	16 MB
`MM`	Matrix Multiplication	AMDAPPSDK	Scatter-Gather	33 MB
`SC`	Simple Convolution	AMDAPPSDK	Adjacent	131 MB
`ST`	Stencil 2D	SHOC	Adjacent	33 MB

B. Baseline Configuration

We conduct our experiments using the industrial-validated MGPUsim Simulator [56]. We target multi-GPU system where each GPU has its own local page table and GPU Memory Management Unit (GMMU). The baseline configurations are shown in Table I. Note that, the GPU memory capacity is configured to a fixed ratio (i.e., 70%) of the application's memory footprint (given in the last column of Table II). This allows us to model the memory oversubscription during execution while avoiding extremely long simulation times when simulating large memory footprints. This approach is also employed by prior work GPU [10], [22], [24], [34]. In the baseline settings, we use a 4KB page size and provide sensitivity to large page size in Section VI-B3. In all the experiments, the thread block (TB) scheduler first schedules the TBs across CUs within one GPU in a round-robin fashion. Only when the GPU cannot accommodate more TBs, the scheduler moves to the next GPU [30], [32]. This scheduling captures the inter-TB locality within a GPU.

IV. Motivation and Characterization

A. Overall Application Characteristics

We plot the page-handling latency of each page placement scheme and normalize the page-handling latency to the baseline on-touch migration. For each scheme, we further break down the page-handling latency into six parts, as shown in Figure 3. Specifically, "Local" captures local page table walk latency after L2 TLB misses. "Host" represents the UVM page faults handling latency. "Page-migration" captures

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

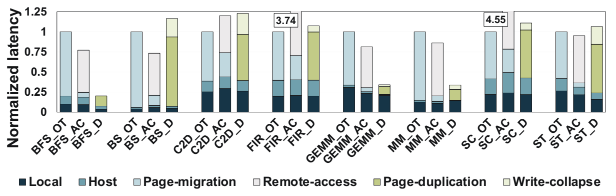


Fig. 3. Page-handling latency breakdown of each page placement scheme. (OT represents on-touch migration, AC indicates access counter-based migration, and D stands for page duplication.)

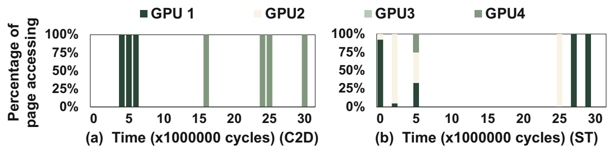


Fig. 5. Shared page access pattern over time for a certain page.

the page migration latency as discussed in detail in Section II-B1. One can observe that "page-migration" accounts for a significant portion of page-handling latency in each application when on-touch migration is employed, whereas, in contrast, the "page-migration" is significantly reduced by allowing remote access when access counter-based migration is employed. However, the counter-based migration introduces a notable increase in remote access latency, which is denoted by the "remote-access" portion in the figure. While "page-migration" and "remote-access" latency are eliminated in the page duplication scheme, duplicating pages introduces two unique latencies labeled as "page-duplication" and "write-collapse". Specifically, "page-duplication" includes latencies of i) UVM driver duplicating page to requested GPU, ii) page eviction from GPU due to oversubscription, and iii) page re-duplication when accessing the evicted page. "Write-collapse" includes latencies of i) the UVM driver walking the centralized page table to obtain the information and ii) the GPU that owns the page flushes in-flight instructions in the CU pipeline, contents of the caches and TLBs, and invalidates the corresponding PTE. In Figure 3, applications `BFS`, `GEMM`, and `MM` show little "write-collapse" latency. This is because these applications are read-intensive, and duplicating shared pages can substantially increase local access. However, "page-duplication" and "write-collapse" can introduce significant latencies in certain applications with frequent reads and writes to the shared pages. As such, those shared pages are frequently collapsed after writes and then re-duplicated after reads by GPUs. For example, in `BS`, `C2D`, and `ST`, it's observed that 46%, 49%, and 45% of their pages, respectively, experience write-collapse being re-duplicated by reads afterward.

B. Page Access Characteristics

Observation 1: The page-sharing patterns vary among different applications and show variations over time within the same application. Figure 4 shows the percentage of private pages and shared pages of each application. We define the *private page* as pages that are only accessed by one GPU during the entire execution, while *shared page* are accessed by more than one GPU during the whole execution. One can observe differences in page sharing across different applications. For example, in `FIR` and `SC`, almost all the pages are private. In contrast, almost all pages are shared by GPUs in `BFS` and `ST`. Applications such as `C2D` and `MM` show a mix of both private and shared pages. The sharing behavior significantly impacts the performance of page placement schemes. For private pages, pages are exclusively accessed by a single GPU. Therefore, migrating the page on touch to the requesting GPU maximizes the locality for subsequent accesses. This is also the reason why on-touch migration outperforms other page placement schemes in `FIR` and `SC` (see Figure 1), where a significant portion of private pages are observed. We also show the percentage of accesses to private pages and shared pages in Figure 4. One can observe that, except `BFS`, all other applications have a majority of page accesses going to the dominating page types (i.e.,g private or shared). While `BFS` has a large number of shared pages, only a few accesses are going to shared pages. Therefore, employing access counter-based page migration is appropriate for these shared pages.

For shared pages, we further investigate their access behaviors during the course of execution. We collect the distribution of the accesses to a page from all GPUs at intervals of one million cycles for `C2D` and `ST`, who have a significant amount of page sharing. As shown in Figure 5, we can classify the shared pages into two categories. First, *producer-consumer shared page (PC-shared page)*, where the page is dedicatedly accessed by one GPU at certain intervals and then accessed dominantly by another GPU at different intervals, as shown in the `C2D` (Figure 5 (a)). For such scenarios, adopting on-touch migration is more favorable. Second, *all shared page*, where different GPUs access the shared page frequently throughout the execution, as shown in the `ST` (Figure 5 (b)). In this case, access counter-based migration is beneficial as this scheme involves tracking the access counts of each shared page on different GPUs. If a particular page is currently residing on a GPU with a low access count, the page becomes a potential candidate for migration to another GPU with a higher access count. We also observe sharing pattern change for the same page in different intervals within `ST`. Specifically, the same page in `ST` shows the all-shared pattern during intervals 0-5 and becomes the PC-shared pattern during intervals 25-30. Therefore, for those applications with extensively shared pages (e.g., `BS`, `C2D`, and `ST`), it is important to dynamically decide the most suitable page placement scheme for different applications and different pages during the executions considering the variations of page sharing patterns.

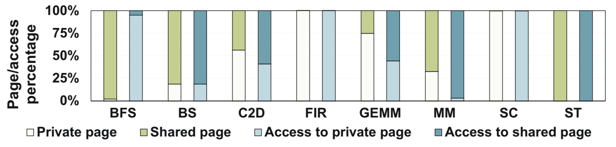


Fig. 4. Percentage of private page and shared page, and percentage of accesses going to private pages and shared pages.

**Observation 2: Page duplication does not always yield

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

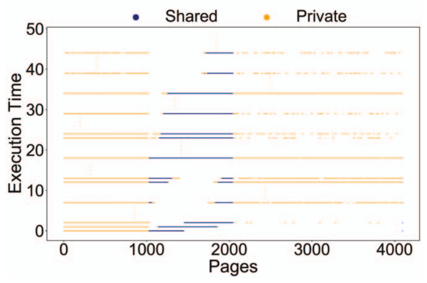


Fig. 6. Private or shared page attribute over time for all pages (GEMM).

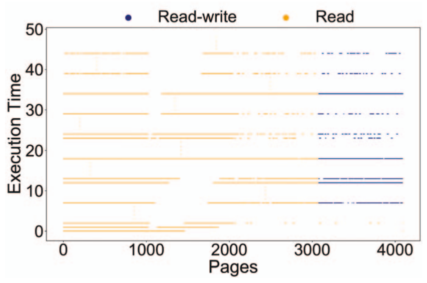


Fig. 7. Read or read-write page attribute over time for all pages (GEMM).

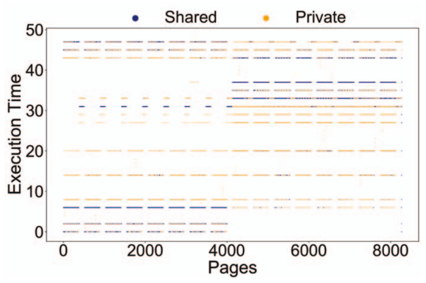


Fig. 8. Private or shared page attribute over time for all pages (ST).

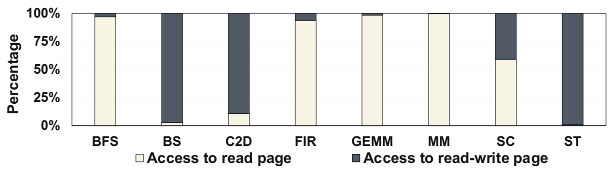


Fig. 9. Percentage of the accesses going to read pages and read-write pages.

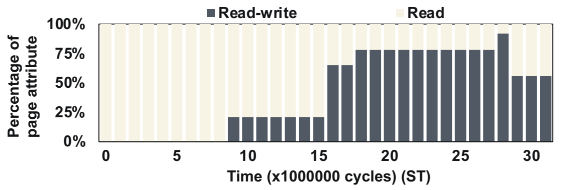


Fig. 10. Read and read-write attribute over time for a certain page.

benefits for read-write intensive applications. Ideally, page duplication can significantly improve performance as all pages can be accessed in local memory when reused. However, as shown in Figure 1, the read duplication does not always yield the best performance. Figure 9 presents this problem using the distribution of GPU memory accesses to read pages and read-write pages. The *read page* refers to a page where all memory accesses to that page are read during the entire execution. The *read-write page* is defined as a page that experiences at least one write operation during the execution. Combining with Figure 1, one can make the following observations. First, the page duplication works well for applications with substantial read-shared pages (e.g., BFS and GEMM), since duplicating the shared pages locally can significantly reduce remote memory access latencies. Second, the page duplication does not provide benefits to applications with intensive read-write pages (e.g., BS, C2D, SC, and ST). This is because the overheads of page write-collapse can be expensive.

Moreover, read or read-write page attributes also exhibit time variations within the same application. To illustrate this, we further study the memory access patterns of the read-write page during the execution. Figure 10 shows the distribution of the read/write memory accesses to a particular read-write page at intervals of one million cycles for ST. We observe that write memory accesses are not always present throughout the execution of the application. Instead, there are intervals where there are only read accesses (intervals 0-8) and other intervals where there are both read and write accesses (intervals 9-31). This highlights the importance of carefully choosing the page duplication, considering the variation in read/write memory access behavior for different pages and different periods.

Takeaway. The study above reveals that different applications benefit from different page placement schemes, indicating that there is no one-size-fits-all scheme. The degree of page sharing and page read/write varies across different applications, which affects the effectiveness of different page placement schemes. Within a specific application, different pages exhibit distinct access patterns, and their behaviors may vary over time. This dynamic nature of page access behavior calls for a dynamic page placement scheme that can accommodate variations in page access characteristics.

C. Page Attributes Characterization

We observe that in many applications, the neighboring pages tend to exhibit similar access attributes. To illustrate it, we sample the accesses attributes (i.e., shared/private and read/write) of consecutive 4,000 pages throughout the entire execution of GEMM, and the results are presented in Figure 6 and Figure 7. To be more specific, the y-axis represents the total execution cycles, and we divide the entire execution cycle into 50 intervals. For each interval, we track all the page access attributes. As observed, for example, at the same time interval, pages 0 to 1,000 exhibit the same private and read attribute, while pages 1,000 to 2,000 display the same shared attribute. This pattern is also closely related to the algorithmic data structure in the application. In the case of GEMM, the algorithm allocates three separately consecutive memory segments for the two input matrices and the output matrix. During execution, each GPU reads data for the input matrices from both its local memory and the remote GPU's memory, then each GPU writes its computed portion of the output matrix only to its own local memory (i.e., DRAM). The output matrix is divided into portions, and each GPU is responsible for computing and updating its assigned portion. As a result, the shared pages, which contain data from the input matrices, remain read and are accessed by multiple GPUs. These pages are accessed consecutively in the matrix, leading to consistent access patterns among neighboring pages within a certain interval. On the other hand, the write pages, which

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

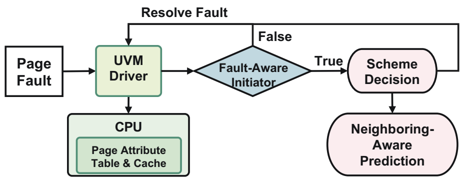


Fig. 11. High level overview of GRIT.

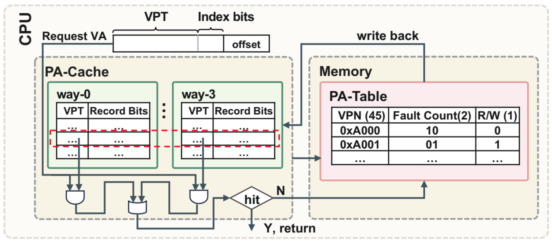


Fig. 12. Overview of PA-Table and PA-Cache in GRIT.

store the output matrix data, are accessed sequentially in the matrix and modified exclusively by a specific GPU, making them private and consecutive. We also present the private and shared page attributes over time of one irregular application (ST) to demonstrate the neighboring page access attributes. We can observe from Figure 8 that, even though the attributes of specific pages change over time, neighboring pages exhibit similar page attributes as well as attribute changes over time. This observation highlights that if we can track one page's access attributes, we can predict the neighboring page access behavior ahead of time, enabling us to proactively determine the page placement scheme for neighboring pages.

V. fine-**GR**ained dynam**I**c page placemen**T** (GRIT)

A. High Level Overview

We propose fine-**GR**ained dynam**I**c page placemen**T** (GRIT) that leverages page access attributes to dynamically determine the page placement scheme at runtime. There are three major challenges in effectively and efficiently determining a page placement scheme. First, different applications have distinct page access behaviors and the page access patterns of an application change over time. Therefore, it is important to promptly identify improper page placement schemes during the execution and determine when to initiate the scheme change. Second, to identify the right moment to change the scheme and select a suitable scheme, it is important to record the application page access attributes without introducing significant overhead. Finally, inaccurate page placement scheme predictions can lead to unnecessary page migration overhead or increased remote memory accesses. Therefore, it is important to accurately predict the neighbor page attributes. To this end, we propose GRIT as shown in Figure 11, which incorporates three novel designs: i) Fault-Aware Initiator to determine when to initiate page placement scheme change, ii) Page Attribute Table to monitor the page access characteristics (i.e., share/private and read/write) to provide information for selecting appropriate page placement schemes, and iii) Neighboring-Aware Prediction to proactively decide neighboring page placement scheme by leveraging the similarity of access patterns and page attributes among consecutive pages.

B. Fault-Aware Initiator

One intuitive way to detect whether the current page placement scheme is suitable is to periodically check the per-GPU and per-page attributes. However, the periodic checks require each GPU to maintain access patterns and attribute information for every page, which results in significant storage overhead in each GPU. Additionally, interconnection communication overhead arises as the information needs to be shared across different GPUs to analyze access patterns and page attributes for each GPU. To mitigate these overheads, we employ the *page fault* as an indicator to trigger the page placement scheme change, which includes the number of local page faults and page protection faults. When a page translation is invalid in the local page table entry, it generates a local page fault that is sent to the CPU for handling. Frequent occurrences of local page faults for the same page indicate that the page is being accessed frequently by multiple GPUs and that the current page placement scheme is unsuitable, suggesting that page duplication may be more appropriate. Additionally, as mentioned in Section II-B3, in the page duplication, the UVM driver will receive a page protection fault when a write operation is performed on a shared page. If write operations frequently occur on a shared page, it indicates that the page duplication is not suitable due to the expensive page write collapsing overheads, thus a scheme change is demanded. Therefore, by monitoring the number of local page faults and page protection faults received in the UVM driver, we can efficiently detect unsuitable page placements without incurring additional storage and interconnection overhead. The default fault threshold is set to four[1] to initiate the scheme change. When a specific page reaches the fault threshold, it triggers an interruption to the UVM driver for scheme change. We implement a Page Attribute Table (PA-Table) in the CPU memory to track the number of page faults for each page. We discuss PA-Table details next and address questions: *How to track page information?* and *Which scheme to change to?*.

C. Page Attribute Table

Page Attribute Table (PA-Table) is designed to indicate page attributes (i.e., private/share and read/write) and track the number of page faults (i.e., local page faults and page protection faults). Figure 12 shows the architecture details of PA-Table. Specifically, each entry in the PA-Table is 48 bits and stores VPN (45 bits), read/write type (1 bit), and fault counter (2 bits, initialized to 00). In this PA-Table design, accessing PA-Table involves additional memory accesses, potentially impacting the memory bandwidth of running applications and introducing additional overheads. To mitigate this issue, we introduce a

[1]We also evaluate our approach with different fault thresholds (i.e., the total number of local page faults and page protection faults) in Section VI-B1.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

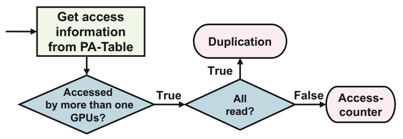
hardware-managed Page Attribute cache (PA-Cache) to store frequently accessed entries, thereby mitigating memory bandwidth contention caused by additional memory accesses to the PA-Table. Figure 12 also illustrates the microarchitecture of PA-Cache. The cache is designed for 64 entries with a 4-way associative structure. The PA-Cache employs a write-allocate and write-back policy.

How to track page attribute: When UVM receives a local page fault or page protection fault, it initiates the page table walk to resolve the fault request and also checks the PA-Cache to obtain the access information of the corresponding page. Specifically, we design 64 entries in the PA-Cache. The VPN is divided into index bits (the lower 4 bits of VPN) and virtual page tag (VPT, the upper bits of VPN excluding the index bits). The index bits of the request are used to locate the set in the PA-Cache. The VPT of the request is then compared with the VPT stored in the corresponding set. If the VPT of the request is found in the PA-Cache, the fault counter is incremented by 1. The read/write bit is set as the requested page attribute (0 for read, 1 for write). Once the read/write bit is set to 1 (write), it remains unchanged during the current scheme lifetime. The entry in the PA-Table will be deleted once the fault counter reaches the fault threshold and the page attribute is updated to a new scheme. However, if the VPT is not found in the PA-Cache, a memory access is generated to access the PA-Table. Two scenarios may happen. If the entry is found in the PA-Table, the corresponding entry is then brought into the PA-Cache as in the write-allocate policy, and the fault counter and read/write bit are updated. Otherwise, the VPT of the request and the corresponding bit are registered in the PA-Cache. The reason for bringing the entry to PA-Cache and updating it there instead of directly updating it to PA-Table is that there is a high possibility for other GPUs to access this page subsequently due to page sharing. If the PA-Cache is full, an entry is evicted using the LRU replacement policy and is written back to the PA-Table. If the fault counter reaches the fault threshold, the page access information is sent to the UVM driver for the following analysis, and both the entry in the PA-Cache and the PA-Table are deleted. Note that, we do not include a bit for private and shared page characteristics. This is because the corresponding entry is deleted after the page placement scheme is changed. In the case of a page accessed by only one GPU after deletion, it will never generate a local page fault or page protection fault. Therefore, when a request successfully hits either PA-Cache or PA-Table and triggers a scheme change, it indicates that the page is a shared page.

TABLE III
POLICY PREFERENCE.

Types	Private	PC-shared	All-shared
Read	OT/Duplication	OT/Duplication	Duplication
Read-write	OT	OT/AC	AC

Which scheme to change: From the characterization in Section IV, we can derive the candidate page placement scheme related to the page attributes, as described in Table III. Based on the page access information obtained from PA-Table,



Get access information from PA-Table
Accessed by more than one GPUs?
True
All read?
True
Duplication
False
Access-counter

Fig. 13. Scheme decision mechanism in GRIT.

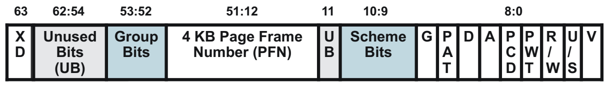


Fig. 14. Page table entry format for 4KB pages in GRIT.

we propose a page placement scheme decision mechanism as shown in Figure 13. If a page placement scheme change is triggered, it indicates that the corresponding page is a shared page. This is because a privately accessed page, which is exclusively accessed by a single GPU, generates only one local page fault and is registered in the PA-Table upon initial access. Afterward, the page is migrated to the local memory, and the translation mapping is established in the local page table. Consequently, private pages do not trigger any updates to the PA-Table, and page placement scheme changes are not initiated for such pages. Therefore, our mechanism simplifies the decision-making process by solely checking the read/write bits of the page. If all accesses to a page are read, the scheme is changed to page duplication. Conversely, if a page is accessed by write, the access counter-based migration is chosen. The adopted page placement scheme will be updated in both the host side PTE and the GPU PTE. The bits of 9 to 10 in the PTE are used to store the scheme bits as shown in Table IV and Figure 14. Note that, there is no need to initiate another page table walk to update the scheme bit. This is because the scheme bit update occurs during the page table walk to resolve the page fault. The PA-Cache and PA-Table lookup and decision-making latency (e.g., PA-Table lookup only needs one memory access) is generally less than page table walk latency (e.g., an average of 2-3 memory accesses depending on page walk cache performance). In situations where the page table walk is faster than making a scheme decision, we hang on the page table walk and let it wait for the scheme decision to be finalized. Note also that, there are potential overheads associated with scheme changes. When the scheme is reset from duplication to another scheme, data consistency needs to be ensured. In such cases, the UVM driver invalidates the corresponding PTE/TLB in each GPU.

TABLE IV
SCHEME BITS.

Scheme Bits	01	10	11
Scheme	On-touch migration	Access counter-based	Duplication

D. Neighboring-Aware Prediction

Recall our discussion in Section IV, neighboring pages tend to exhibit similar page attributes. We further propose a Neighboring-Aware Prediction approach, which leverages attribute similarity to predict the attributes of neighboring pages and proactively determine the page placement scheme. It allows pages that have not yet been allocated in the GPU's

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

page table or are still using their previous page placement scheme to apply the new scheme on the next page fault occurrence without having to reach the fault threshold. Specifically, we define a page group as a set of consecutive pages (i.e., pages are located adjacently in the virtual address space). The minimum page group size is eight 4KB pages, and eight smaller groups can be combined to form a larger group. In our approach, we set the maximum group size to 512 pages, which corresponds to a 2MB continuous virtual address space that can be accommodated within a single page table page. This design eliminates the need to check across different page table pages and avoids generating additional memory accesses. We leverage the unused bits in the host side PTE to create a group size bit (i.e., bits 52 to 53, as shown in Figure 14), which identifies the number of consecutive pages within the page group. For example, if bits 52 and 53 are set to “00”, it indicates a single 4KB page. If the bits are set to “01”, it represents a page group of eight consecutive 4 KB pages. The mapping of group bits and their corresponding number of pages is illustrated in Table V. Note that, the reason for leveraging PTE records this information instead of embedding it in the PA-Table is that the virtual addresses of consecutive pages are stored sequentially in the page table. When we know the virtual address of the base page[2] in the group and the size of the group, we can easily identify all the consecutive pages within that group by simply traversing the page table entries sequentially. The group size bits are only recorded in the PTE of the base page in each page group to simplify the process of managing and analyzing consecutive page groups. The virtual address of the base page (VPN_{base}) is calculated as follows: $VPN_{base} = VPN_{curr} - (VPN_{curr}/PageSize)\%GroupSize \times PageSize$. It is important to note that if the group size is set to “01” or larger, it indicates that all the pages within the group adopt the same page placement scheme.

TABLE V
GROUP BITS AND THEIR CORRESPONDING NUMBER OF PAGES.

Group bits	Number of pages	Size
00	1	4KB
01	8	32KB
10	64	256KB
11	512	2MB

As shown in Figure 15, initially, all group size bits are set to “00”. When the number of page faults of a specific page reaches the page fault threshold, the page placement scheme change is initiated (❶). Once a new scheme is determined for this particular page, we then check the page placement scheme (bits 9 to 10 of the PTE) for eight neighboring pages (❷). If more than half of these checked pages adopt the same page placement scheme as the newly selected scheme for this specific page, we apply this new scheme to all these pages and update the corresponding scheme bits for each page. Then, we promote these eight pages into a group and update group bits of the base page to “01” to indicate all eight consecutive

[2]We define the first page in each group as the *base page*.

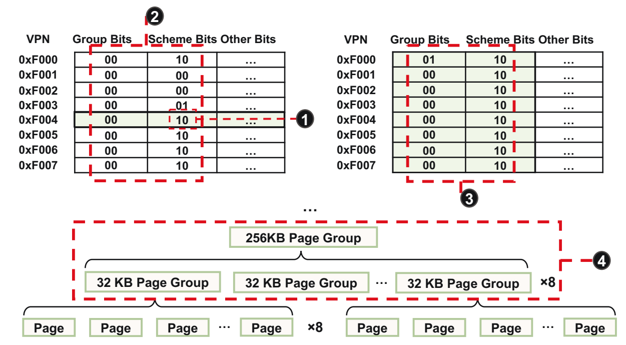


Fig. 15. Neighboring-Aware Prediction in GRIT.

pages now adopt the same page placement scheme and can be treated as a cohesive unit (❸). We then recursively check if this group can combine with other neighboring groups to further promote to a larger group. Similarly, if more than half of the neighboring groups (i.e., the group size bit of the base page is “01”) adopt the same page placement scheme as the newly determined scheme, we further propagate the scheme to all these pages (i.e., 64 pages), and update the group size bit of the base page to “10”, indicating that 64 pages are now adopting the same page placement scheme (❹).

When the page fault of a specific page within the group reaches the fault threshold and initiates a scheme change, the scheme bit of this page is changed to the new scheme, making it different from the scheme used by other pages within the same group. As a result, we will perform a downward degradation of the page group, wherein the original group will be downgraded to a smaller group due to the presence of a newly selected different scheme within the group. For example, if the group bits are initially “10” (indicating a group size of 64 consecutive pages employing the same page placement scheme), and one of the pages within the group changes to another scheme, the 64-page group is degraded into eight 8-page groups. The group of pages where the scheme change occurred will change group bits to “00” because one of these eight pages is now using a different scheme and cannot be considered a unified group anymore. However, the other seven 8-page groups derived from the initial group still use the same scheme, and hence, their group bits are set to “01”.

Note that, when the newly determined page placement scheme is the same as the previously adopted page placement scheme, which can only occur in the access counter-based migration, we do not perform page group checks to prevent unnecessary back-and-forth group promotion and degradation. For example, consider a page group with eight pages, all employing the access counter-based migration. When one page within the group changes to the duplication, the group is downgraded to single eight pages. Subsequently, the second and third pages also change to duplication, and upon checking the neighboring eight pages, only three pages use the same duplication, which is insufficient to trigger a group promotion. Then, if another page within this neighboring range initiates a scheme change and the decision remains the same as the

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

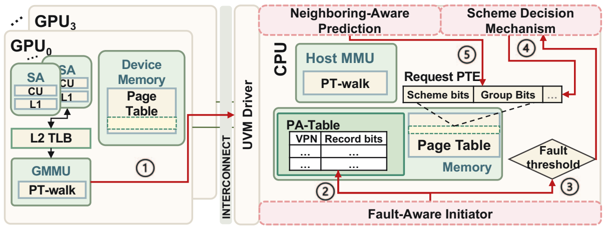


Fig. 16. Overview of GRIT.

previous scheme, i.e., access counter-based migration, it will not perform a page group check. This is because more than half of the neighboring eight pages are still using the access counter-based migration scheme, and if we choose to promote to a larger group, the three pages that employ duplication will be changed back to the access counter-based scheme. Avoiding page group checks in such cases prevents unnecessary group promotion and degradation.

Note also that, the page group check happens in the background so that page placement scheme updates do not block GPU execution. Once the local page fault or page protection fault is resolved, the GPU resumes its execution. In the background, the UVM driver checks if the page placement scheme can be unified for neighboring pages, and this process does not involve any page migration or PTE/TLB invalidation.

E. Putting All Together

Figure 16 provides an overview of the entire process of GRIT. When a read request is generated, and it misses the L1, L2 TLB, and local page table, the GMMU sends a local page fault to the host (the GMMU sends a page protection fault for a write request) (①). Upon receiving the page fault, the UVM driver updates the PA-Table and PA-Cache to record and check the fault information (②) in parallel with page table walks. Depending on the number of faults that have occurred for this specific page (③), two scenarios may happen. First, if the fault has not reached the threshold, the UVM driver checks the scheme bits of the PTE in the centralized page table when performing the page table walk. If the scheme bits are different from the scheme currently being employed due to the neighboring-aware prediction, the page employs the updated scheme determined by Neighboring-Aware Prediction as specified in the centralized PTE without waiting to reach the fault threshold. Second, if the fault count has reached the fault threshold, the UVM driver uses the access information stored in PA-Table to decide the appropriate scheme to be applied (④). It then updates the scheme bits in PTE for both the GPU and host side accordingly. Also, the UVM driver triggers Neighboring-Aware Prediction to further optimize for neighboring pages and update group bits (⑤).

F. Overheads

Our proposed GRIT introduces two main overheads. First, the PA-Table incurs memory overheads and the PA-Cache incurs hardware overheads. In our design, each entry in the PA-Table is 48 bits (45 bits for VPN + 2 bits for page fault + 1 bit for read/write attribute) for a single page. Given that the page size is 4KB, the total memory space required is $\frac{48bits}{4KB} = 0.15\%$ of the application memory footprint. Thus, the memory overhead of the PA-Table is negligible compared to the overall memory in the system. With 64 entries in the PA-Cache, the hardware overhead is $(41 + 2 + 1)\ bits \times 64\ entries = 352\ bytes$. We use CACTI [58] to estimate the areas and the results show that PA-Cache is 0.04% compared to the areas of 32KB 8-associative CPU L1 cache. Second, scheme change involves latency overheads. It can happen that a new scheme decision is made after the page table walk has finished. In such cases, the replay of page fault is postponed until the scheme bit is updated in the page table entry, and this can impact the total latency of page fault handling. However, we rarely observe such case happens in our evaluation and this additional latency is marginal to the overall performance. When the scheme is reset from duplication to another scheme, the UVM driver removes all the page replicas and invalidates the corresponding PTE and TLB in each GPU to ensure data consistency. Although this process introduces latency, it is considered trivial compared to the overheads caused by improper schemes.

VI. Evaluation

A. Overall Performance

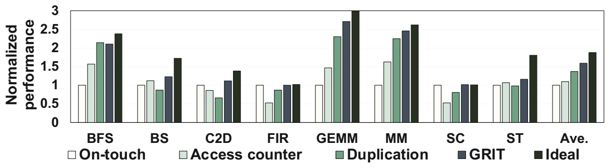


Fig. 17. Performance of each scheme relative to baseline on-touch migration.

We evaluate our proposed approach using the same benchmarks in Table II. The baseline architecture configuration is identical to Table I. Figure 17 plots the performance of GRIT and the three page placement schemes (i.e., on-touch migration, access counter-based migration, and page duplication). The results are normalized to the baseline on-touch migration. GRIT achieves an average of 60%, 49%, and 29% performance improvements compared to uniformly employing on-touch page migration, access counter-based migration, and page duplication, respectively. The performance benefits stem from the effectiveness of capturing different page access patterns and configuring appropriate schemes for different pages and different applications, as we elaborate in detail next. First, our approach is able to capture different preferred page placement schemes for different applications. For example, in `BFS`, our scheme achieves similar performance compared with page duplication as the majority of the pages in `BFS` are read pages. In contrast, `FIR` and `SC` prefer on-touch migration as most of the pages are private and our approach can capture that behavior and adjust the page placement scheme accordingly. It is important to note that, the slight performance drop (2%) in `BFS` is because our design starts with on-touch migration as the baseline and gradually adjusts the page placement schemes, which involves generating GPU local page fault. This

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

incurs additional overheads compared to uniformly using page duplication at initialization. Second, our approach effectively captures the different page placement schemes during the execution of a given single application. For example, in GEMM and MM, page duplication is better compared to on-touch migration and access counter-based migration as approximately half of the pages are shared, whereas the other half of pages are private in these two applications. Our approach still achieves 17% and 9% performance improvements over page duplication as it is able to capture the read-write pages. That is, our approach achieves the improvement for GEMM and MM by fine-tuning the page placement for read pages to duplication and optimizing the private read-write pages to on-touch migration. Third, for ST, BS, and C2D, while GRIT achieves the highest performance improvement over three schemes, there is still a large gap between GRIT and the ideal performance. This is because ST, BS, and C2D have a significant amount of shared read-write pages (99%, 56%, and 42%, respectively) over the entire execution, making any migration schemes less effective.

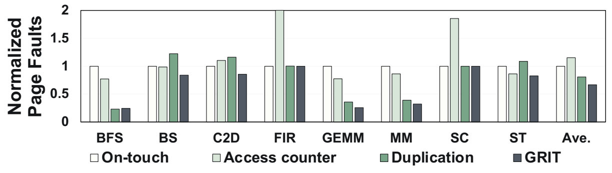


Fig. 18. The number of page faults.

The number of GPU page faults (including both local page faults and page protection faults) is closely correlated with the performance results, as these faults can significantly degrade overall performance due to frequent UVM handling and CPU interruption. If the page placement scheme is properly determined, it ensures that most pages are present in local memory or page table entries are valid in the local page table. It also avoids frequent page write collapsing, resulting in fewer local page faults and page protection faults. To help understand the performance improvements, we present the total number of GPU page faults when employing different page placement schemes and GRIT in Figure 18. The results are normalized to the number of page faults in on-touch migration execution. GRIT achieves 39%, 55%, and 16% reduction in the total number of GPU page faults compared to on-touch migration, access counter-based migration, and page duplication, respectively. This clearly demonstrates the effectiveness of our approach in adapting to various access patterns and efficiently determining the appropriate page placement schemes.

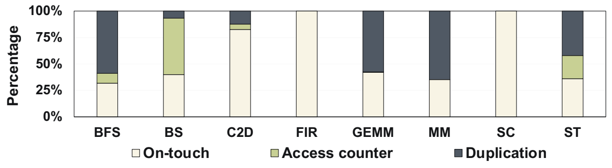


Fig. 19. Percentage of each page placement scheme by using GRIT.

We further demonstrate the effectiveness of our approach by plotting the breakdown of the page placement scheme during the whole execution. Specifically, Figure 19 shows the percentage of different page placement schemes among all the accesses that miss the GPU L2 TLBs. Note that, a single page can change its schemes multiple times during different execution phases. Each of the scheme changes is captured in the figure. We observe a hybrid pattern in many applications. For BFS, GEMM and MM, page duplication is the predominant choice due to the substantial read shared pages, which matches the performance results that duplication yields the best performance across all three page placement schemes. For C2D, the on-touch migration is most commonly employed. The primary reason for this is that the majority of shared pages in C2D are shared by two GPUs, following a producer-consumer pattern (as shown in Figure 5). This pattern only occurs two page faults, which fall below the fault threshold. Therefore, the scheme continues to employ the initial on-touch migration. This sharing pattern also reflects the performance results, where the on-touch migration gains the highest performance in C2D across three schemes. For BS, access counter-based migration is primarily employed due to the substantial number of all-shared pages. The primary choice of access counter-based migration by GRIT matches the performance results, in which access counter-based migration achieves the best performance across all three schemes. For ST, duplication and on-touch migration play an important role due to the read pages during a certain interval and PC-shared read-write pages. The choice of duplication and on-touch migration by GRIT matches the performance results of ST, where these two schemes outperform access counter-based migration. For FIR and SC, the on-touch migration remains the most proper scheme since almost all the pages are private as characterized in Figure 4. In a nutshell, our approach is able to distinguish page attributes and consistently select the most suitable scheme accordingly.

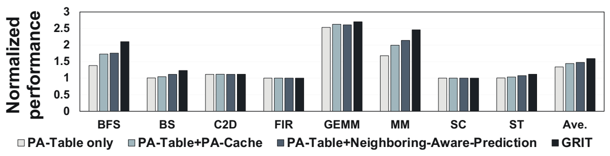


Fig. 20. Performance of different components in GRIT.

Figure 20 shows performance improvements of each individual component of GRIT (i.e., PA-Table only, PA-Table+PA-Cache, and PA-Table+Neighboring-Aware-Prediction) normalized to baseline on-touch migration scheme. The results show that PA-Table only, PA-Table+PA-Cache, and PA-Table+Neighboring-Aware-Prediction achieve an average of 31%, 47%, and 44% performance improvement over the baseline, respectively. This demonstrates the effectiveness of each GRIT component, and these components collaboratively enhance the performance.

B. Sensitive Study

1) Different fault thresholds: Recall that we leverage a fault threshold to trigger the scheme optimization. A larger threshold indicates that more GPU local page faults are needed to trigger page scheme change, thereby delaying the

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

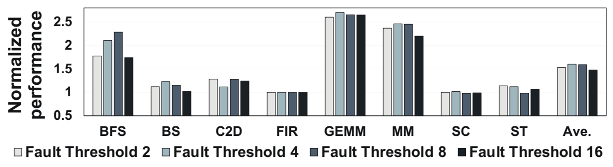


Fig. 21. GRIT using 2, 4, 8, and 16 as the fault threshold.

effectiveness of timely capturing page access patterns. In contrast, a smaller threshold leads to frequent and potentially "ping-pong" page placement scheme changes, increasing the overheads in search and adjusting the policies. To choose an appropriate fault threshold, we show the performance results employing different thresholds (i.e., 2, 4, 8, and 16) normalized to baseline on-touch migration in Figure 21. The performance improvements over the baseline on-touch migration are 53%, 60%, 59%, and 48%, to the fault thresholds of 2, 4, 8, and 16, respectively. As one can observe, the performance gain saturates when employing 4 as the fault threshold. Therefore, we choose 4 as the fault threshold in our reported main results.

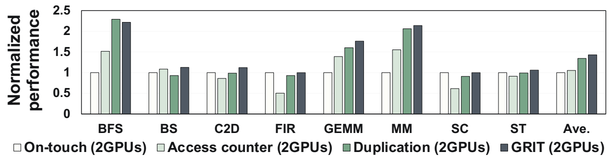


Fig. 22. Performance with 2 GPUs.

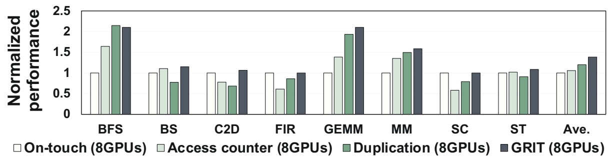


Fig. 23. Performance with 8 GPUs.

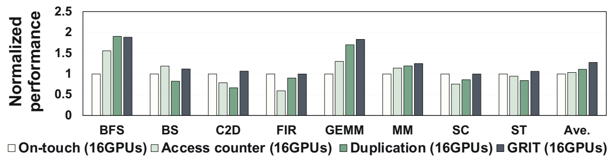


Fig. 24. Performance with 16 GPUs.

2) Different number of GPUs: We also evaluate GRIT in 2-GPU, 8-GPU, and 16-GPU systems. Figure 22, Figure 23, and Figure 24 present the performances of GRIT with 2 GPUs, 8 GPUs, and 16 GPUs normalized to baseline with 2 GPUs, 8 GPUs, and 16 GPUs, respectively. Note that, in this experiment, we only change the number of GPUs and keep the same application input size for a fair comparison. For 2 GPUs, GRIT achieves 40%, 37%, and 11% performance improvements over on-touch migration, access counter-based migration, and page duplication, respectively. The performance improvements are 38%, 35%, and 26% in 8 GPUs and 27%, 26%, and 23% in 16 GPUs, respectively. To understand the performance improvements, we also quantify the page fault reduction achieved by our approach. With 2 GPUs, we observe reductions of 34%, 42%, and 11% compared to on-touch migration, access counter-based migration, and page duplication, respectively. In 8 GPUs, the reductions are 31%, 45%, and 19% for the same schemes. In 16 GPUs, the reductions are 30%, 47%, and 15% for the same schemes. The page fault reduction is similar to the 4-GPU system (39%, 55%, and 20%), which demonstrates our approach remains effective with different numbers of GPUs. Note that, the decreased performance improvement as the number of GPUs increases does not mean a weakening of the effectiveness of our approach. This is because the pages become more frequently shared among GPUs with more GPUs, leading to more page migrations. Regardless of the page placement scheme adopted, page migration latency is unavoidable, and as the number of migrations increases, the overall impact of this latency becomes dominant in the total execution time. This leads to a diminishing in performance improvement as the potential benefits gained from improved page placement are reduced by the increased overhead caused by frequent page migrations.

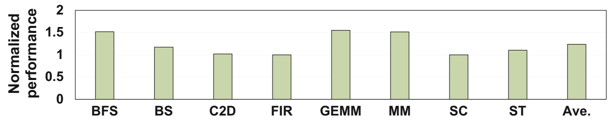


Fig. 25. GRIT with 2MB pages.

3) Large page: We evaluate GRIT with 2MB page. Note that, to sufficiently stress the virtual memory subsystem, we enlarge the application input size (i.e., the memory footprints span from 0.5GB to 3GB). The result is shown in Figure 25, and the performance is normalized to the baseline with 2MB page size and large input sizes. The average performance improvement is 23% compared to the baseline on-touch migration. GRIT maintains its effectiveness even when adopting larger pages. However, the performance improvement (23% comparing GRIT-2MB to baseline-2MB) is reduced (60% comparing GRIT-4KB to baseline-4KB). This is because 2MB introduces more frequent false sharing among GPUs. Consequently, the page attributes become mixed. For example, in a sequence of 512 consecutive 4KB pages, there are both read pages and read-write pages. We can utilize page duplication for read pages and access counter-based migration for read-write pages. However, when these pages are merged into a larger 2MB page, the page attributes become read-write, and we can only use the access counter-based migration, resulting in more remote memory accesses.

C. Compared to State-of-the-art

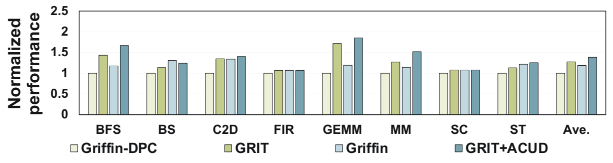


Fig. 26. Comparison of Griffin-DPC, GRIT, Griffin, and GRIT+ACUD.

1) Comparison to Griffin: We compare GRIT with the state-of-the-art multi-GPU page migration management, i.e.,

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

Griffin [10]. Griffin has two major components: i) Dynamic Page Classification (DPC) which classifies pages into different categories and decides which pages to migrate and ii) Asynchronous Compute Unit Draining (ACUD) which reduces the overhead of pipeline draining and flushing when pages are migrated. Figure 26 compares the performances of Griffin-DPC, GIRT, Griffin, and GRIT+ACUD normalized to Griffin-DPC. We implement DPC using the same default hyper-parameter configurations as Griffin. We make the following observations. GRIT achieves 27% performance improvement over Griffin-DPC (the first two bars). The reasons are two-fold. First, Griffin-DPC triggers page migration at a predefined time interval, resulting in substantial remote accesses before the page migration. In contrast, our Neighboring-Aware Prediction can effectively predict the neighboring page attributes and pre-determine an optimal page placement scheme, which significantly reduces the remote memory access caused by improper page placement schemes. Second, Griffin-DPC periodically tracks access information on each GPU, which introduces significant communication overheads between CPU and GPUs. In contrast, GRIT tackles this issue by tracking access information on the CPU side, where the scheme change latency can be hidden during page table walks without introducing additional latency. Next, we implement ACUD on top of GRIT (GRIT+ACUD) and compared it with Griffin with DPC and ACUD (Griffin). Comparing GRIT and GRIT+ACUD, we observe a 9% average performance improvement since ACUD is orthogonal to GRIT. Also, the last two bars in Figure 26 indicate that GRIT+ACUD achieves 16% improvement over Griffin, indicating that GRIT yields additional benefits over Griffin when ACUD is employed.

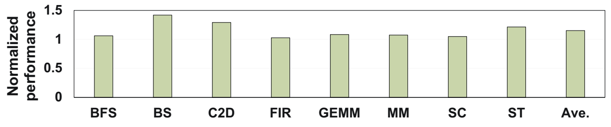


Fig. 27. Comparison to GPS [38].

2) *Comparison to GPS:* We also compare GRIT with the state-of-the-art peer-to-peer data access: GPS [38]. GPS automatically tracks the subscribers (i.e., the GPUs accessed shared page) to each page of memory and proactively broadcasts fine-grained stores to these subscribers, enabling each subscriber to read data from their local memory at high bandwidth. Note that, we implement GPS with the same GPS structure size as described in the original paper, while the GPU configurations follow our GPU parameters as listed in Table I. Figure 27 presents the performance of GPS and GRIT, normalized to GPS. GRIT achieved 15% improvement compared to GPS. This is because GPS duplicates a physical replica in the GPU's local memory once this GPU accesses a page, for applications (e.g., MM, BS, ST) with a majority of shared page accesses during the whole execution, almost all pages will be duplicated in each GPU, which leads to severe memory oversubscription. The detailed memory oversubscription modeling is discussed in Section III-B. We monitor and track memory usage and page eviction events during the runtime. The results indicate that GPS has an average of 34% higher page oversubscription rate compared to our approach. This introduces extra overheads and performance penalties.

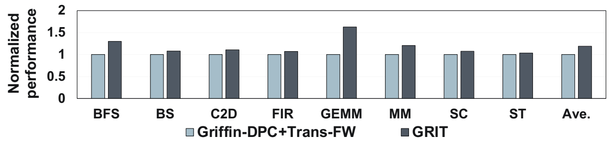


Fig. 28. Comparison to Griffin-DPC [10] combined with Trans-FW [32].

3) *Comparison to the combination of prior works:* We compare GRIT with the combination of Griffin-DPC and Trans-FW [32]. The Griffin-DPC aims to reduce the number of page migrations while Trans-FW concentrates on minimizing the overhead associated with handling page faults caused by page migration. These two approaches are orthogonal. As shown in Figure 28, GRIT achieves an average of 18% improvement compared to the combination. This is because GRIT enables more local accesses and reduces the number of page migrations.

D. Comparison to First-Touch Migration

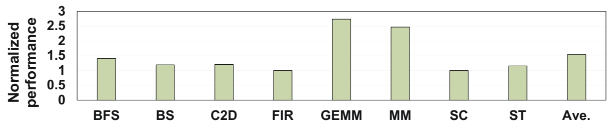


Fig. 29. Comparison to first-touch.

We compare GRIT to first-touch migration, Figure 29 shows GRIT achieves an average of 54% performance improvement. First-touch migration pins the page on the GPU where that page is first accessed and uses peer-access for page sharing across GPUs. It works well for applications with a majority of private page accesses (e.g., `FIR` and `SC`), and GRIT achieves marginal improvements over first-touch. However, it suffers remote access overheads for applications with a majority of shared-page accesses (e.g., `MM` and `GEMM`), where GRIT achieves significant performance improvements.

E. Combine with Prefetching

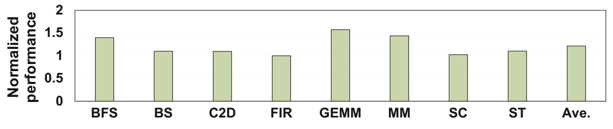


Fig. 30. Performance of GRIT combined with prefetching [23].

Prefetching is a technique that leverages the data locality to proactively fetch data from the remote device's memory to the local memory before it is needed. Ganguly et al. [23], [24] revealed a tree-based neighborhood prefetching approach implemented in NVIDIA CUDA driver [45]. Specifically, the system maintains a set of full-binary trees, where the leaf levels hold 64KB basic blocks, and the root nodes correspond to 2MB page addresses. The runtime continuously monitors the current memory occupancy for each GPU and maintains

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

this information for each node in the binary trees. When the runtime detects that a specific GPU's occupancy of a non-leaf node in the tree data structure exceeds 50% of the node's total capacity, it selects the leaf nodes under that node as prefetch candidates and fetches them to that GPU. We next combine GRIT with the tree-based neighborhood prefetching approach. Figure 30 shows that GRIT with prefetching approach achieves 23% performance improvement over the baseline on-touch migration with the prefetching approach. This is because our approach is able to proactively determine the page placement scheme, which is complementary to the prefetching approach and brings additional performance benefits.

F. DNN workloads

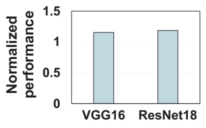


Fig. 31. DNN.

We evaluate GRIT using VGG16 and ResNet18 model parallelism on multi-GPUs. Figure 31 shows that GRIT achieves 15% on VGG16 and 18% on ResNet18 performance improvements over their baseline executions. This indicates that GRIT also works in multi-GPU-based DNN training.

VII. Related Work

NUMA Optimization: Substantial prior studies have focused on improving the performance of NUMA systems [1], [10], [11], [19], [27]–[29], [31]–[33], [36], [61], [63]. Agarwal et al. [1] developed an intelligent data migration mechanism for GPU-CPU systems. Young et al. [63] proposed to improve NUMA-GPU by caching remote data in video memory (CARVE), which dedicates a small fraction of the GPU memory to store the contents of remote memory. Different from these NUMA optimizations, our approach focuses on efficient page placement by fine-tuning different page placement schemes for pages at runtime, which is complementary to most of the multi-GPU NUMA optimization.

Runtime Page Placement: Previous research has explored methods aimed at enhancing the performance of page placement schemes [1]–[3], [19], [35], [62]. Dashti et al. [19] presented a memory management algorithm that leverages interleaving, page replication, and page migration, which addressed the traffic congestion issue and mitigated the cost of remote wire delays. Agarwal et al. [3] proposed Thermostat, a mechanism for placing pages in a hybrid memory system at runtime that detects and acts on hot and cold pages. However, none of these efforts dive into runtime page attributes, based on which further improve the page placement. In our work, we entail a comprehensive analysis of page attributes, leveraging these insights to further optimize runtime page placement for multi-GPU NUMA systems.

VIII. Conclusion

In this paper, we proposed GRIT which dynamically determines page placement schemes in a fine-grained manner to enhance multi-GPU page placement. Specifically, we propose Fault-Aware Initiator to detect the inappropriate page placement scheme, Page Attribute Table to track the page attribute and to determine the optimal page placement scheme, and Neighboring-Aware Prediction to proactively determine the page placement scheme by predicting page attributes for adjacent pages. Experimental results show that our proposed GRIT achieves an average of 60%, 49%, and 29% performance improvements compared to uniformly employing on-touch migration, access counter-based migration, and page duplication, respectively.

Acknowledgements

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

References

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for gpus in cc-numa systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 354–365.

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

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

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

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

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

[7] AMD. (2017) AMD Radeon™ Instinct™ MI25 Accelerator. [Online]. Available: https://www.amd.com/en/products/professional-graphics/instinct-mi25

[8] N. Amit, "Optimizing the {TLB} shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 27–39.

[9] N. Amit, A. Tai, and M. Wei, "Don't shoot down tlb shootdowns!" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.

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

[11] L. Belayneh, H. Ye, K. Chen, D. Blaauw, T. Mudge, R. Dreslinski, and N. Talati, "Locality-aware optimizations for improving remote memory latency in multi-gpu systems," in *2022 31th International Conference on Parallel Architectures and Compilation Techniques*, 2022.

[12] C.-H. Chang, A. Kumar, and A. Sivasubramaniam, "To move or not to move? page migration for irregular applications in over-subscribed gpu memory systems with dynamap," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, 2021, pp. 1–12.

[13] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

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

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

[15] S. Choi, I. Koo, J. Ahn, M. Jeon, and Y. Kwon, "{EnvPipe}: Performance-preserving {DNN} training framework for saving energy," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 851–864.

[16] Y. Dai, X. Tang, and Y. Zhang, "FlexGM: An Adaptive Runtime System to Accelerate Graph Matching Networks on GPUs," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 348–356.

[17] Y. Dai, Y. Zhang, and X. Tang, "CEGMA: Coordinated Elastic Graph Matching Acceleration for Graph Matching Networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 584–597.

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

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

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

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

[22] D. Ganguly, R. Melhem, and J. Yang, "An adaptive framework for oversubscription management in cpu-gpu unified memory," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1212–1217.

[23] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 224–235. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322224

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

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

[26] H. Jiang, Y. Chen, Z. Qiao, T.-H. Weng, and K.-C. Li, "Scaling up mapreduce-based big data processing on multi-gpu systems," *Cluster Computing*, vol. 18, pp. 369–383, 2015.

[27] A. K. Johansen, "Fast multi-gpu communication over pci express benchmarking pcie transport with the nvidia collective communications library (nccl) using legacy gpus," Master's thesis, 2023.

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

[29] J. Lee, J. M. Lee, Y. Oh, W. J. Song, and W. W. Ro, "Snakebyte: A tlb design with adaptive and recursive page merging in gpus," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1195–1207.

[30] B. Li, Y. Guo, Y. Wang, A. Jaleel, J. Yang, and X. Tang, "IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1163–1177.

[31] B. Li, Y. Wang, and X. Tang, "Orchestrated scheduling and partitioning for improved address translation in gpus," in *In Proceedings of the 60th Design Automation Conference (DAC)*, 2023.

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

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

[34] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 49–63. [Online]. Available: https://doi.org/10.1145/3297858.3304044

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

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

[37] S. P. Mohanty, "Gpu-cpu multi-core for real-time signal processing," in *2009 Digest of Technical Papers International Conference on Consumer Electronics*, 2009, pp. 1–2.

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

[39] H. Muthukrishnan, D. Lustig, O. Villa, T. Wenisch, and D. Nellans, "Finepack: Transparently improving the efficiency of fine-grained transfers in multi-gpu systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 516–529.

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

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

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

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

[44] NVIDIA Corp. (2016) Nvidia pascal architecture. [Online]. Available: https://www.nvidia.com/en-us/data-center/resources/pascal-architecture-whitepaper/

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

[46] T. Pany, D. Dötterböck, H. Gómez-Martínez, M. S. Hammed, F. Hörkner, T. Kraus, D. Maier, D. Sánchez-Morales, A. Schütz, P. Klima *et al.*, "The multi-sensor navigation analysis tool (musnat)–architecture, lidar, gpu/cpu gnss signal processing," in *Proceedings of the 32nd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2019)*, 2019, pp. 4087–4115.

[47] E. Park, J. Ahn, S. Hong, S. Yoo, and S. Lee, "Memory fast-forward: A low cost special function unit to enhance energy efficiency in gpu for big data processing," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 1341–1346.

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

[49] M. M. Rathore, H. Son, A. Ahmad, A. Paul, and G. Jeon, "Real-time big data stream processing using gpu with spark over hadoop ecosystem," *International Journal of Parallel Programming*, vol. 46, pp. 630–646, 2018.

[50] L. Savioja, V. Välimäki, and J. O. Smith, "Audio signal processing using graphics processing units," *Journal of the Audio Engineering Society*, vol. 59, no. 1/2, pp. 3–19, 2011.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.

[51] D. Schaa and D. Kaeli, "Exploring the multiple-gpu design space," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[52] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 2013, pp. 73–82.

[53] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, and M. Guo, "Oversubscribing gpu unified virtual memory: Implications and suggestions," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 67–75.

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

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

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

[57] R. Tang, Z. Zhao, K. Wang, X. Gong, J. Zhang, W. Wang, and P.-C. Yew, "Ascetic: Enhancing cross-iterations data efficiency in out-of-memory graph processing on gpus," in *Proceedings of the 50th International Conference on Parallel Processing*, 2021, pp. 1–10.

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

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

[60] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward unified-memory-efficient high-performance graph processing on gpu," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 2, pp. 1–25, 2021.

[61] C. Xie, F. Xin, M. Chen, and S. L. Song, "Oo-vr: Numa friendly object-oriented vr rendering framework for future numa-based multi-gpu systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 53–65. [Online]. Available: https://doi.org/10.1145/3307650.3322247

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

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

[64] Y. Zhang, D. Peng, X. Liao, H. Jin, H. Liu, L. Gu, and B. He, "Large-graph: An efficient dependency-aware gpu-accelerated large-scale graph processing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–24, 2021.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 13,2024 at 14:22:29 UTC from IEEE Xplore. Restrictions apply.