2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

# OASIS: Object-Aware Page Management for Multi-GPU Systems

Yueqi Wang[1], Bingyao Li[1], Mohamed Tarek Ibn Ziad[2], Lieven Eeckhout[3],
Jun Yang[1], Aamer Jaleel[2], Xulong Tang[1]
University of Pittsburgh[1], NVIDIA[2], Ghent University[3]
yuw249@pitt.edu, bil35@pitt.edu, mtarek@nvidia.com, lieven.eeckhout@ugent.be,
juy9@pitt.edu, ajaleel@nvidia.com, tax6@pitt.edu

***Abstract*—The ever-growing need for high-performance computing has driven the popularity of employing multi-GPU systems. Modern multi-GPU systems employ unified virtual memory (UVM) to manage page placement and migration. However, the page management in UVM is application object agnostic. In this paper, we characterize the page access behaviors in relation to the application objects, and reveal that the beneficial page management policy varies according to (i) the different data objects within the same application, and (ii) the different execution phases of the same object. This motivates the need for dynamic and proactive page management in multi-GPU systems. To this end, we propose OASIS, which dynamically identifies object patterns during the execution and proactively determines the appropriate page management policies for these objects at runtime. Experimental results show that OASIS improves the performance over uniformly adopting on-touch migration, access counter-based migration, and duplication by an average of 64%, 35%, and 42%, respectively. Moreover, OASIS achieves a 12% performance improvement over the state-of-the-art technique (i.e., GRIT) while having significantly lower design complexity.**

## I. Introduction

Multi-GPU systems have emerged as the preferred platform to meet the ever-growing demands of high-performance computing, offering enhanced parallelism and expanded memory capacity [15], [19], [20], [24]–[26], [33], [38], [39], [51], [55], [57]. Modern multi-GPU systems generally employ Unified Virtual Memory (UVM) [37], [42], which simplifies application development by avoiding manual memory management across devices. In addition, UVM's runtime memory management accommodates evolving GPU configurations and memory capacities. UVM also broadens the application of multi-GPU systems to fields with irregular and unpredictable memory access patterns, such as graph processing, where data cannot be easily and statically partitioned across GPUs. Despite its advantages, UVM-enabled multi-GPU systems suffer from Non-Uniform Memory Access (NUMA) overheads that arise from data sharing and communication across GPUs.

Modern UVM-enabled multi-GPUs utilize three different page management policies to mitigate NUMA overheads: (i) on-touch page migration [41], (ii) access counter-based migration [42], and (iii) page duplication [42]. Specifically, the on-touch policy always migrates pages to the requesting GPU's local memory, enabling high bandwidth access for all subsequent accesses to the same page. However, it causes frequent page migrations when GPUs frequently share the page. To address this, the access counter-based migration policy only migrates pages when page accesses reach a certain threshold (e.g., 256 remote accesses in NVIDIA UVM driver [37], [40]), alleviating frequent migrations but incurring remote access latency before a page is migrated [3], [4], [7], [8]. The page duplication policy enables concurrent local reads by duplicating pages but requires invalidating all copies upon a write to the page (called page write-collapse) [22], [23], [35].

Current multi-GPU systems uniformly employ one of the aforementioned page management policies. However, each of these policies has distinct pros and cons, and there is no universal solution that serves good for all applications. This variability arises because applications often contain different types of data, each favoring a different page management policy. For example, on-touch migration is beneficial for data that is accessed exclusively by one GPU, enabling subsequent accesses to be served locally. In contrast, if data is shared among multiple GPUs, on-touch migration can incur substantial migration overheads. These can be significantly reduced by employing access counter-based migration. Furthermore, if the shared data is predominantly read-only, duplication proves to be the most effective solution, ensuring that all shared read accesses are served locally.

Several prior studies [11], [53], [56] propose optimizations to the three policies. For example, the state-of-the-art multi-GPU approach GRIT [53] dynamically learns the beneficial management policy on a per-page basis. Unfortunately, learning memory access patterns on a per-page basis can incur considerable memory overheads and additional memory access latency. Moreover, GRIT [53] incorporates hardware prediction for neighboring page management, which this study finds to be overly complex and unnecessary. We aim for a simpler and more efficient memory management design compared to GRIT and show quantitative evidence (Section VI-C).

While making page management decisions at the page granularity can be implemented in the operating system and hardware, ensuring transparency to applications, we observe that operating at such a fine granularity overlooks opportunities for more efficient page management at a reduced cost. To this end, we choose to strike a balance between application transparency and page management efficiency by leveraging *object* granularity. An object is a data structure that is dynamically created at runtime by calling a memory allocation

function, such as `cudaMallocManaged`. Tracking objects at runtime provides an efficient and transparent way for memory management as object access patterns naturally represent the application memory page access behavior. Making memory management decisions at the object granularity, which is typically coarser than page granularity, eliminates the overhead of learning a policy for each individual page, as a single object typically comprises a significant number of pages.

We show that preferred page management decisions can be determined based on the sharing characteristics of an object. For example, the on-touch policy is optimal for an object that is only accessed by a single GPU. Similarly, the page duplication policy is beneficial for a read-only object that is accessed by multiple GPUs. Finally, the access-counter policy is suitable for a read-write object that is accessed by multiple GPUs. Based on these insights, we propose OASIS, which captures the object runtime access pattern and dynamically adapts the most suitable memory management policy at the object granularity. Specifically, OASIS consists of three key components: (1) the Object Tracker to identify objects during run-time, (2) the Object Table to capture the sharing pattern of objects based on access patterns, and (3) the Object Policy Controller to dynamically modify the page management policy based on the observed object access patterns.

We make the following contributions in this paper:

1) Our characterization of page management at object granularity reveals several key insights: (i) pages allocated to the same object exhibit similar access patterns; (ii) objects demonstrate varying preferences for page management policies; and (iii) object patterns remain consistent within a specific execution phase but may change at the phase boundary.
2) To the best of our knowledge, OASIS is the first work to leverage object information for enhancing page management decisions in multi-GPU systems. We propose hardware-assisted OASIS, an object-aware page management system designed to dynamically identify object patterns and determine the most effective page management policy for each object at runtime. Additionally, we design a software-only alternative, OASIS-InMem, to accommodate diverse demands.
3) We evaluate OASIS using 11 representative multi-GPU applications. Results show that OASIS improves the overall performance by 64%, 35%, and 42% over uniformly adopting on-touch migration, access counter-based migration, and duplication, respectively. It also outperforms the state-of-the-art GRIT [53] by 12% on average and delivers near-optimal performance with much lower design complexity.

## II. Background

### A. UVM-Enabled Multi-GPU Memory Management

In this paper, we focus on UVM-enabled multi-GPU systems, where multiple GPUs are connected via high-bandwidth interconnects such as PCIe [36] or NVLink [21]. The unified virtual memory (UVM) [21] is employed in multi-GPU systems and is managed by the UVM driver located on the CPU. UVM greatly simplifies GPU application development by establishing a shared memory space between CPUs and GPUs. This shared memory is facilitated by a crucial function called `cudaMallocManaged` [43]. This function allocates a continuous block of virtual memory for a specified data object, making it universally accessible. As a result, both the CPU and GPU can use universal pointers to directly access this shared memory space, which eliminates the need for manual data transfers. Figure 1 shows the GPU architecture of the baseline multi-GPU system. Each GPU has its own local memory and local page table. The GPU Memory Management Unit (GMMU) handles GPU local page table walks. The UVM driver maintains a centralized page table on the host CPU side that holds up-to-date translations from all connected GPUs and CPU. When a kernel running on a GPU accesses a page where its page table entry is invalidated in the local page table, it triggers a page fault. The page fault is then sent to the UVM driver and is resolved using the centralized page table.

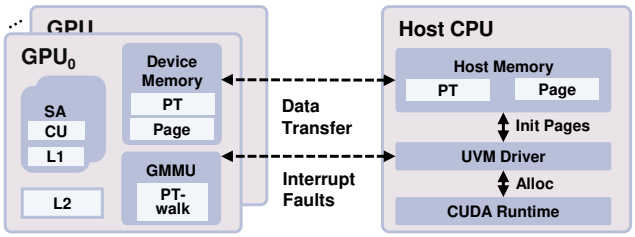



Fig. 1. UVM-enabled multi-GPU architecture.

### B. Page Management Policy

The currently adopted page management policies in multi-GPU infrastructures are (i) on-touch page migration, (ii) access counter-based page migration, and (iii) page duplication.

*1) On-touch migration:* This policy always migrates a page into the local memory of the requesting GPU. While it benefits all subsequent accesses of the same GPU to the same page (as they become local memory accesses), it causes frequent "ping-pong" page migrations when multiple GPUs frequently share pages. This significantly degrades performance because page migration is expensive and introduces execution overheads.

*2) Access counter-based migration:* NVIDIA Volta GPUs and newer generations [13], [37] introduce a policy known as access counter-based migration. This policy allows a GPU to access pages stored in the memory of a remote GPU by establishing the address translation of the remote page in its local page table. The access counter-based migration utilizes a hardware-based access counter to track the number of remote accesses. Page migration only occurs when a certain threshold is met (e.g., 256 remote accesses for a 64 KB page group [37], [40]). This policy resolves the "ping-ponging" of pages under on-touch migration; however, it incurs remote access latency before a page is migrated and page table entry (PTE) invalidation overheads [32].

*3) Page duplication:* The page duplication policy duplicates read-shared pages among GPUs and allows concurrent

TABLE I
BASELINE MULTI-GPU CONFIGURATION.

| Module | Configuration |
|---|---|
| Compute Units | 1.0 GHz, 64 per GPU |
| L1 Vector Cache | 16 KB, 4-way |
| L1 Inst Cache | 32 KB, 4-way |
| L1 Scalar Cache | 16 KB, 4-way |
| L2 Cache | 256 KB, 16-way |
| DRAM | 4 GB |
| L1 TLB | 32 entries, 32-way, CU private, LRU replacement policy |
| L2 TLB | 512 entries, 16-way, CUs shared, LRU replacement policy |
| Access counter threshold | 256 [40] |
| Inter-GPU network | 300 GB/s NVLink-v2 |
| CPU-GPU network | 32 GB/s PCIe-v4 |

local reads to duplicated pages. However, when a GPU performs a write operation, it sends a *page protection fault* to the UVM, and all shared pages in other GPUs have to be invalidated to ensure consistency. The page duplication allows read-shared pages to be accessed by multiple GPUs locally, avoiding remote memory accesses. However, the overhead of collapsing read-write shared pages can lead to significant performance degradation [17], [35], [42], [56].

## III. METHODOLOGY

### A. Baseline Configuration

We conduct our experiments and evaluations using the industry-validated MGPUSim Simulator [49]. Our experimental setup includes a 4-GPU platform, where each GPU has its local page table and GMMU. Additionally, we extend our experiments to analyze performance across varying numbers of GPUs: in Section VI-B2, we provide a sensitivity study where OASIS is evaluated on platforms with 8 and 16 GPUs. Our baseline configurations are detailed in Table I. These settings include the use of a standard 4 KB page size, and we further study the impact of using a large page size in Section VI-B4.

### B. Applications

We use eleven applications from well-known benchmark suites, including AMDAPPSDK [5], Hetero-Mark [48], SHOC [16], and DNN-MARK [18], as detailed in Table II. These applications span a broad spectrum of domains, including machine learning, graph algorithms, and numerical computations. Moreover, these applications cover various multi-GPU access patterns as listed in Table II, enabling a thorough analysis of multi-GPU access patterns and their implications on system performance. Specifically, `BFS` and `PR` demonstrate random access patterns, with GPUs performing read and write operations to and from different GPUs unpredictably. `C2D`, `ST`, `LeNet`, `VGG16`, and `ResNet18` exhibit an adjacent access pattern, where input data is batched and sequentially shared among neighboring GPUs. `I2C`, `FFT`, `MM`, and `MT` incur a scatter-gather access pattern, with each GPU handling data from local or remote GPUs. Note that data parallelism is employed for these DNN workloads by distributing the data across multiple GPUs. The `LeNet` model utilizes the MNIST dataset [30], a large collection of handwritten digits (0-9) with 70,000 images, split into 60,000 for training and 10,000 for testing. The dataset used for `VGG16` and `ResNet18` is Tiny-Imagenet-200 [54], which contains 100,000 images distributed across 200 classes, with each class consisting of 500 training images, 50 validation images, and 50 test images. We also list the number of objects for each application under the "# Objects" column in the table. Note that the number of objects reported here represents the maximum number of objects allocated throughout the execution. The actual number of objects utilized during any specific period may be fewer. We also report the memory footprint in Table II, ranging from 20 MB to 300 MB, which is sufficient to capture the memory access patterns in multi-GPU systems [11], [53]. Note that simulating large memory footprints is impractical due to extremely long simulation times. The memory footprint sizes are similar as in prior multi-GPU studies [11], [53].

TABLE II
LIST OF APPLICATIONS.

| Abbr. | Application | Benchmark Suite | Access Pattern | # Objects | Memory Footprint |
|---|---|---|---|---|---|
| `BFS` | Breadth-First Search | SHOC | Random | 5 | 32 MB |
| `C2D` | Convolution 2D | DNN-Mark | Adjacent | 10 | 92 MB |
| `FFT` | Fast Fourier Transform | SHOC | Scatter-Gather | 2 | 48 MB |
| `I2C` | Image to Column | DNN-Mark | Scatter-Gather | 3 | 80 MB |
| `MM` | Matrix Multiplication | AMDAPPSDK | Scatter-Gather | 4 | 32 MB |
| `MT` | Matrix Transpose | AMDAPPSDK | Scatter-Gather | 3 | 64 MB |
| `PR` | Page Rank | Hetero-Mark | Random | 6 | 32 MB |
| `ST` | Stencil 2D | SHOC | Adjacent | 3 | 32 MB |
| `LeNet` | LeNet | DNN-Mark | Adjacent | 115 | 24 MB |
| `VGG16` | Visual Geometry Group 16-layer | DNN-Mark | Adjacent | 240 | 220 MB |
| `ResNet18` | Residual Network 18-layer | DNN-Mark | Adjacent | 263 | 297 MB |

## IV. MOTIVATION AND CHARACTERIZATION

### A. Overall Application Characteristics

We plot the performance of uniformly employing a single page management policy on all pages in Figure 2, in which the performance numbers are normalized to the on-touch migration (i.e., baseline policy). The "Ideal" bar represents an ideal NUMA-GPU system that duplicates all shared pages, irrespective of read or write operations. Specifically, all initial accesses to pages from a GPU (i.e., both read and write accesses) incur a duplication latency and duplicate that page to the requesting GPU. For subsequent accesses to these pages, once they are already present locally, they are treated like read accesses, incurring zero NUMA latency regardless of whether the operation is a read or a write. It is important to note that this "Ideal" configuration is hypothetical and not feasible in practice; it serves merely to illustrate the potential for optimization. The results in the figure highlight that there is no universal page management policy that consistently achieves the best performance across various applications. This is because object access patterns vary across different applications and execution phases within the same application.

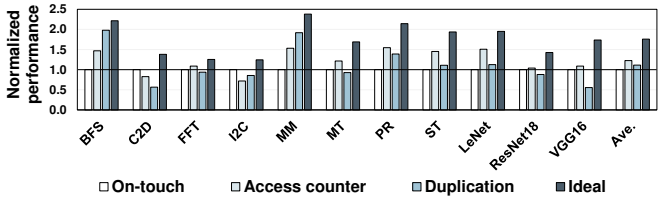



Fig. 2. Performance of different page-management policies normalized to the baseline on-touch migration policy.

**Observation 1**: No single page management policy fits all applications because data access patterns vary across different applications.

### B. Object Characteristics

To optimize page management policies for various applications, we begin by conducting a detailed analysis of memory access patterns during multi-GPU execution.

**Basic Terminology.** To facilitate the discussion in this section, we first introduce the following terms to categorize memory access patterns of pages: (i) private pages: pages accessed exclusively by one GPU during a particular execution period; (ii) shared pages: pages accessed by more than one GPU within a specific period; (iii) read-only, write-only, or read-write-mix (rw-mix) pages: pages that are only read from, only written to, or both read and written to by the GPUs during a certain execution period. We then define the memory access patterns of an object by analyzing the patterns of its pages. Specifically, if all pages or over 90% of the pages exhibit a consistent pattern, we classify the object according to this predominant pattern. For instance, if all pages within $\text{object}_0$ are private, we identify this object as a private object. If pages within an object show a variety of patterns, we categorize the object with a 'mix' pattern. For example, if 30% of an object's pages are read-only and 70% are write-only or rw-mix, we categorize this object as an rw-mix object. Similarly, a private-shared-mix object contains a combination of both private and shared pages. Additionally, since the patterns of private/shared and read/write are orthogonal, we can combine them to describe the object pattern. For example, a shared-read-only object indicates that multiple GPUs access the object and it is only read by those GPUs.

**Effectiveness of Object Granularity.** We demonstrate the effectiveness of focusing on object granularity by characterizing the distribution of object sizes across our evaluated applications, as shown in Figure 3. One can observe that while the smallest objects may consist of only a single 4 KB page, most objects tend to have larger sizes, spanning multiple pages. In addition, our evaluation reveals that pages within an object tend to exhibit consistent behavior. For example, Figure 4 illustrates the access patterns of `MT`, where the x-axis represents pages. The first approximately 9,000 pages, belonging to MT_Input, are entirely read-only, while the next 9,000 pages, belonging to MT_Output, are all write-only. It is worth noting that scaling `MT` to a larger input size (e.g., 2 GB) does not affect the object count or page access patterns.

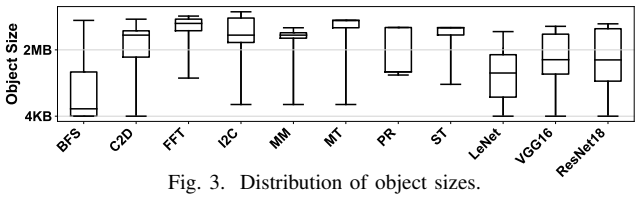



Fig. 3. Distribution of object sizes.

In this case, MT_Input and MT_Output would encompass more pages (i.e., 262,144 each), with MT_Input read-only and MT_Output write-only. This underscores the effectiveness of object-level analysis, allowing us to focus on two objects, each with pages exhibiting consistent patterns, instead of analyzing over 500,000 pages.

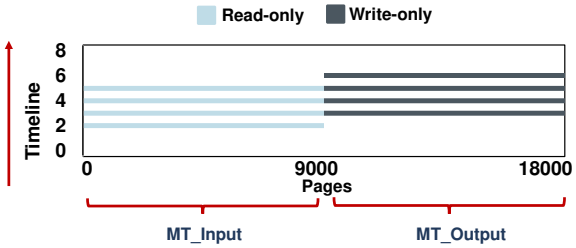
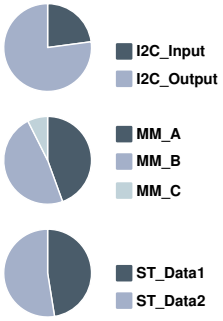



Fig. 4. The page access patterns related to objects of MT.

It is important to note that this uniformity in page patterns within the same object is also observed in other benchmarks besides `MT`, though not all objects show 100% consistency across their pages. To quantify this, we introduce the concept of a *non-uniform object*, defined as having at least one page whose pattern differs from others of this object in both dimensions (i.e., private/shared, read/write). We then define the concept of a *non-uniform app*, where at least one object within the application is non-uniform. We evaluate 7 applications that only have one explicit phase (i.e., kernel function) throughout their application execution (i.e., `BFS`, `FFT`, `I2C`, `MM`, `MT`, `PR`, and `ST`). Note that, in this paper, we define *phase* as an execution period during which the objects exhibit distinct, recognizable patterns. We further categorize the phase into two types: explicit and implicit. Explicit phases, such as CUDA kernels, are identifiable by the runtime system at the time of launch. Implicit phases, in contrast, are not explicitly recognizable but exhibit recognizable pattern shifts during execution, usually associated with iterative processes. Our evaluation shows that only 2 out of 26 objects are non-uniform, and only 1 out of 7 applications (i.e., `ST`) qualify as non-uniform apps, with non-uniform pages comprising less than 5% of the total. We exclude applications with multiple explicit phases from this analysis because their object patterns vary across phases, leading to them being classified as non-uniform applications. Even in this non-uniform application `ST`, our subsequent analysis reveals that its object behavior exhibits consistency during implicit phases. These pattern variations will be thoroughly studied in subsequent parts of this section.

**Observation 2**: Pages within a single object typically exhibit the same patterns. We can use object granularity to study access patterns.

| I2C | Object Pattern | # Pages |
|---|---|---|
| I2C_Input | Shared-read-only | 2048 |
| I2C_Output | Private-write-only | 18379 |
| **MM** | **Object Pattern** | **# Pages** |
| MM_A | Shared-read-only | 4096 |
| MM_B | Shared-read-only | 2048 |
| MM_C | Private-write-only | 2048 |
| **ST** | **Object Pattern** | **# Pages** |
| ST_Data1 | Shared-rw-mix | 4133 |
| ST_Data2 | Shared-rw-mix | 4129 |




(a) (b)

Fig. 5. (a) Object behavior analysis for three applications. (b) Percentage of accesses going to different objects.

To understand the performance differences of each policy across different applications and reveal the object preference for different page management policies, we analyze the object behavior of three applications. First, as illustrated in Figure 5, I2C primarily consists of two objects: I2C_Input and I2C_Output. I2C_Output, a private object, occupies a larger number of pages and is accessed more frequently (i.e., 75% of the total accesses) compared to I2C_Input, making it play a dominant role in determining performance. This observation, combined with the performance results in Figure 2, one can find that the on-touch migration policy delivers optimal performance compared to the other two policies. This is because on-touch migration exhibits the lowest latency for private objects, as it promptly places the pages to the GPUs that request them. In contrast, access counter-based migration defers data migration until the counter threshold is met, leading to increased remote access latency. Note that when using duplication, a read-only copy of the page is made on the requesting GPU. Even if a page is private, any attempt to write to the read-only copy will still trigger a page protection fault, introducing significant fault handling overhead. This prevents duplication from achieving optimal performance for private-rw-mix objects. Second, as shown in Figure 5, MM comprises three primary objects: MM_A, MM_B, and MM_C. Among these, MM_A and MM_B, are shared-read-only and account for the majority of pages as well as the accesses (i.e., 80% of the total accesses). Recall the performance result in Figure 2 showing that the duplication policy demonstrated the best performance for MM. Duplication performs best for shared-read-only objects by maintaining local replicas, thus minimizing remote data access and data migration. Third, as shown in Figure 5, ST owns two main objects: ST_Data1, and ST_Data2, both of which are shared-rw-mix. The performance result in Figure 2 illustrates that access counter-based migration delivers better performance for ST over on-touch and duplication. This is because on-touch migration may introduce excessive unnecessary data migrations, and duplication struggles with the overhead from frequent write-collapses. In contrast, access counter-based migration efficiently caters to the GPU that most actively demands the data, as it prioritizes data migration based on request frequency, ensuring that the GPU frequently accessing certain data reaches the threshold

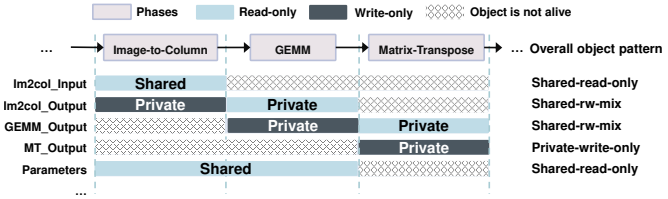



Fig. 6. The object access patterns across different phases of C2D.

first. Consequently, access counter-based migration results in fewer unnecessary migrations than on-touch migration and reduces the write-collapses more efficiently than duplication.

In addition, we want to emphasize that on-touch migration and duplication can achieve the ideal performance for private and read-only objects, respectively. However, although access counter-based migration also shows the best performance among the three policies for shared-rw-mix and shared-write-only objects, it cannot achieve the ideal target. This is because the ideal scenario assumes that all shared pages are duplicated, as detailed in Section IV-A. Achieving the ideal target for shared-rw-mix objects is impossible due to the unavoidable migration latency of on-touch migration, remote access latency of access counter-based migration, or write invalidation of duplication. In contrast, for private objects and read-only objects managed by the on-touch migration and duplication policies, respectively, after the initial cold migration or duplication, subsequent accesses are local, aligning closely with the ideal scenario. Therefore, if we could further partition the shared-rw-mix object into either private or read-only objects across different execution periods, the potential for performance improvement would be maximized. To explore this possibility, we conduct a thorough analysis of object pattern variations.

**Observation 3**: Different objects prefer specific page management policies: private objects favor on-touch migration; read-only benefits from duplication; shared-write-only and shared-rw-mix objects prefer access counter-based policy.

**Characteristics of Object Pattern Variation.** First, we examine the access patterns of objects during a specific explicit phase. Figure 4 also shows the memory access patterns over time. The y-axis represents the total execution time of MT, divided into 8 intervals. One can observe that the object pattern remains consistent throughout the execution of this MT phase. Specifically, the MT_Input remains a read-only pattern, and the MT_Output remains a write-only throughout the entire execution. This consistency arises because, during this specific matrix transpose execution phase, pages belonging to the same object are requested by GPUs in a similar way.

Then, we analyze how their patterns vary across different explicit phases. We plot the object pattern analysis for C2D in Figure 6. We list three main explicit phases of C2D, Image-to-Column, GEMM, and Matrix-Transpose, and we present five objects that dominate these three phases, including Im2col_Input, Im2col_Output, GEMM_Output, MT_Output, and Parameters. The "overall object pattern" refers to the object pattern throughout the entire execution. We mark the

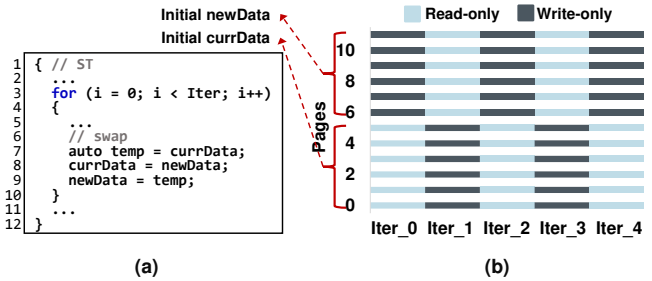



Fig. 7. (a) An example of part of the ST code. (b) The page access patterns across different iterations in ST.

read-only or write-only with different colors, and the private or shared patterns are directly labeled. First, focusing on the "overall object pattern" depicted in Figure 6, we observe that objects such as Im2col_Output and GEMM_Output are shared-rw-mix. However, when tracking the access patterns through various phases (i.e., Image-to-Column, GEMM, and Matrix-Transpose), we find that these objects exhibit a private pattern, with clear distinctions between read-only and write-only patterns across phases, rather than a mixed pattern. Note that an object might be classified as shared throughout the execution but appear as private within individual phases (e.g., Im2col_Output). This occurs when the object is accessed privately by one GPU in a given phase and then exclusively accessed by a different GPU in another phase.

However, we observe that even in some applications without multiple explicit phases, their objects still exhibit regular pattern changes within a single explicit phase. For example, as shown in Figure 7 (a), ST involves two objects, ST_currData and ST_newData, and operates through one Stencil2D function, which includes multiple execution iterations in a loop. Both objects exhibit a shared-rw-mix pattern over multiple iterations throughout the execution. We then study and reveal the object patterns across multiple iterations as illustrated in Figure 7 (b). One can observe from this figure that the total 12 pages are categorized into two parts according to their patterns. The first six pages start as read-only and then switch to write-only, continuing this interleaved pattern of changes. Conversely, the latter six pages start as write-only and exhibit the opposite read/write patterns compared to the first six pages. This behavior occurs because ST reads from ST_currData and writes to ST_newData during each iteration. After this, ST swaps these two objects and proceeds to the next iteration, causing the observed alternations in access patterns. We identify each iteration in the ST explicit phase as an implicit phase. Note that the implicit and explicit features of a phase depend on the algorithm's implementation; different implementations may yield varying implicit and explicit features for the phases. For example, in our evaluated ST, where different iterations are implemented within the kernel function, we categorize the phases implicitly. In a different ST implementation where each iteration would launch a separate kernel, ST would exhibit explicit phases at kernel boundaries.

**Observation 4**: Object access patterns change with phase transitions but remain consistent within the same phase.

**Takeaway.** Our study highlights that different data objects benefit from specific page management policies, indicating that adopting a universal policy is ineffective. The diversity in object patterns across applications influences the performance of different page management policies. Even though memory access patterns for a single object remain consistent within a specific phase, they can change across different phases. The dynamic nature of object patterns, varying both across applications and phases, necessitates the development of flexible and uniquely-designed page management policies.

## V. OASIS Design

### A. System Overview

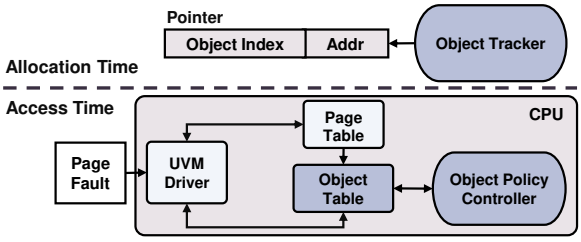



Fig. 8. High-level overview of OASIS.

We propose OASIS and show the overview of our design in Figure 8. OASIS has three major components: (i) an Object Tracker that identifies objects at the time of allocation, using existing allocation APIs (i.e., `cudaMallocManaged()`), and encodes an object ID into the upper pointer bits. This process enables the system to accurately associate memory accesses with their corresponding objects during runtime; (ii) an Object Table that tracks object access patterns during application execution to provide information for page management policy decision-making; and (iii) an Object Policy Controller that leverages the information stored in the Object Table to dynamically adjust page management policies for objects. Given that object patterns may vary across phases, as discussed earlier in Section IV-B, the Object Policy Controller also facilitates automatic runtime corrections, capturing changes in pattern and reassigning the proper policy accordingly.

### B. Object Tracker

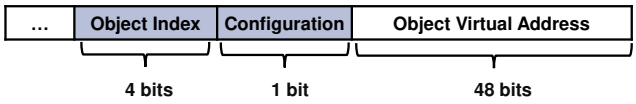



Fig. 9. Pointer layout in OASIS.

To effectively understand the access patterns of objects during different phases, it is crucial to distinctively separate each object. This separation ensures that the CPU and GPUs accurately identify the relevant object during memory access. To achieve this, we propose to encode the object index (Obj_ID) into the unused bits of the pointer that points to this object, as depicted in Figure 9. The number of unused upper pointer bits can vary depending on the host CPU architecture typically ranging from 7 to 16 bits; for our purposes, we assume 5 unused bits. In our design, we use 4 bits to encode Obj_ID and reserve 1 bit for the configuration bit. Note that the

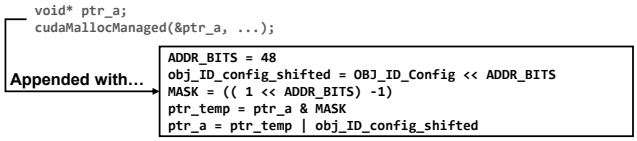



Fig. 10. An example of how to encode object ID into the pointer.

maximum number of bits that can be used to encode Obj_ID is 15. We use only 4 bits as most of the applications we evaluated have fewer than $2^4$ objects. The configuration bit is used to differentiate OASIS and OASIS-InMem (i.e., "1" for OASIS, "0" for OASIS-InMem). OASIS-InMem is designed as a software-only alternative (Section V-F).

The Object Tracker creates a wrapper around the existing memory management APIs (e.g., `cudaMallocManaged()`) to encode the Obj_ID and configuration bit in the upper pointer bits whenever a new object is created. Figure 10 illustrates the pointer encoding process. The memory allocator first calculates `obj_ID_config_shifted` by shifting the Obj_ID and configuration bits left by the addressable bit width (48 bits), positioning them in the upper unused bits of a 64-bit pointer. A mask (`MASK`) is then created to preserve the lower 48 bits of an original pointer `ptr_a`, ensuring any pre-existing higher bits are cleared. Finally, `ptr_a` is reconstructed by combining these extracted lower bits with the shifted Obj_ID and configuration bits, using a bitwise OR operation. The Obj_ID is initialized based on the order of allocation. For instance, the first allocated object is assigned the ID "0000", the second "0001", and so forth. We leverage the Top Byte Ignore feature, which is available on NVIDIA GPUs and modern CPUs, such as ARM's Top Byte Ignore (TBI) [10], Intel's Linear Address Masking (LAM) [27], and AMD's Upper Address Ignore (UAI) [6], to efficiently encode the Obj_ID in the unused upper pointer bits without causing segmentation faults upon pointer dereferencing.

### C. Object Table

The Object-Table (O-Table) in OASIS is an on-chip LRU-managed structure designed to track and record information about each object, such as Obj_ID, and the assigned page management policy. The maximum capacity of the O-Table is fixed at $2^4$. When entries exceed this capacity (e.g., more than 4 unused bits of the pointer are employed for encoding objects), it employs LRU to manage the entries. The O-Table enables the system to (i) facilitate OASIS's Object Policy Controller in learning and assigning appropriate policies during execution, and (ii) record the policy for objects, allowing related memory accesses to directly retrieve the corresponding policy for that object page. The detailed architecture of the O-Table is illustrated in Figure 11. Specifically, each entry within the O-Table occupies 12 bits, comprising a 4-bit Obj_ID, a 1-bit policy bit (0 for duplication, 1 for access counter-based migration), a 3-bit page fault counter (PF Count), and a 4-bit LRU bits. The Obj_ID stores the object index, which is the same as the Obj_ID stored in the pointer. Although the number of entries in the O-Table is fixed, the number of bits for Obj_ID in the O-Table is not fixed but matches the Obj_ID

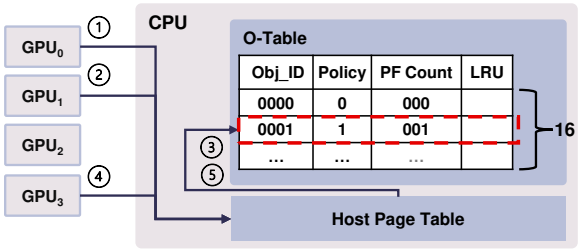



Fig. 11. O-Table overview with an Object Policy Controller example.

bits in the pointer (i.e., up to 15 bits). The policy bit specifies the page management policy applied to pages belonging to this object. We record only duplication and access counter-based migration in the O-Table because on-touch migration is set as the default, and the Object Policy Controller of OASIS only determines between these two policies. We use two unused bits in the page table entry (PTE) to reflect all three policies, as illustrated in Figure 12. This enables both the CPU and GPUs to effectively identify the policy to employ. Specifically, "00" refers to on-touch migration and is set as default, "01" is for access counter-based migration, and "11" indicates duplication. The PF Count tracks the number of page faults forwarded to the O-Table, which is used to monitor the effectiveness of the policy adopted. The detailed process of runtime policy determination is discussed in Section V-D. When an object is allocated, its corresponding entry in the O-Table is initialized. This includes setting the Obj_ID the same as the Obj_ID encoded in the pointer and initializing both the policy bit and PF Count to "0" and "000", respectively. When an object is freed, its entry is removed from the table. As shown in Table II, most of our evaluated applications have fewer than $2^4$ objects. Even in applications with a large total number of objects, only a handful of objects are referenced during a specific phase. As a result, the O-Table requires only a minimal number of entries, with each entry occupying just 12 bits. This modest space requirement allows us to store the O-Table directly in the host CPU's hardware, significantly reducing access overhead.

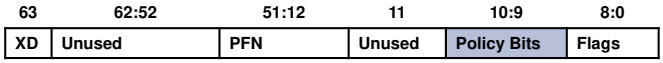



Fig. 12. Page table entry format for 4 KB pages in OASIS.

### D. Object Policy Controller

We use an Object Policy Controller (OP-Controller) that leverages the pattern similarity among same-object pages, enabling dynamic determination of the object policy at runtime. Initially, on-touch migration is set as the default policy for all page handling, with policy bits in the PTE set as "00". The on-touch migration is chosen as the default policy because it allows the CPU to effectively capture memory access patterns by triggering a page fault for every remote request. In contrast, access counter-based migration handles a specified number of remote accesses directly from a remote GPU without sending a page fault to the CPU, and duplication does not trigger a page fault for read-only duplications. In our proposed OP-Controller, we utilize two "tables" to support policy decision-making: the host page table and the O-Table.

**Host Page Table.** We use the host page table to serve private objects. This means that all first-time page faults to a page are processed at the host page table using the default on-touch migration without being forwarded to the O-Table. Accesses to the private object are served only using the host page table because for private objects, the default on-touch migration is already the most effective policy, therefore, it is unnecessary to leverage the O-Table to determine a better policy. Note that identifying whether pages of an object are private or shared does not require additional mechanisms; the host page table naturally provides information on whether pages are private or shared. Specifically, since the physical addresses assigned to different GPUs and the host CPU are typically distinguished by specific physical address ranges, by checking the address range associated with the physical address of the page, the UVM driver can identify which GPU currently holds the data or the data is in the CPU. The data being located on another GPU indicates that this page was accessed by a different GPU previously, thereby confirming that the page is shared. The data in the CPU is considered private as it has not been previously accessed by other GPUs. Note that, oversubscription may influence the classification of data as private or shared. We discuss and evaluate oversubscription in Section VI-D. The host page table acts like an O-Table filter, effectively distinguishing between private and shared objects while also eliminating unnecessary O-Table accesses, thereby avoiding unnecessary memory access overhead.

**O-Table.** We use O-Table to serve shared objects. This means that all subsequent page faults on a page will be forwarded to the O-Table for handling after the host page table lookup. Using the O-Table to handle shared objects allows the OP-Controller to (i) determine the appropriate policy, and (ii) record this policy in the O-Table so that later accesses to the shared object can utilize it. Given that all requests propagated to the O-Table are for shared objects, and the read/write pattern is immediately evident from the request (i.e., the "W" bit in the error code [44]), the OP-Controller obtains sufficient information to decide the policy for the object. Specifically, if the page is read from, the policy bits in the O-Table are set to "0" for duplication; if the page is written to, we set the policy bits to "1" for access counter-based migration. Then the page fault is resolved using this newly employed policy, updating the policy bits in the PTEs for this page to reflect the new policy. Subsequent page faults can directly check the O-Table and employ the policy. We utilize the PF Count in the O-Table to determine whether an incoming request is used to establish a new policy (i.e., PF Count is "000") or employs an existing policy (i.e., PF Count is not "000").

**Example.** We illustrate the process of the OP-Controller with an example, as shown in Figure 11. Assume $GPU_0$ accesses $page_1$ of object $obj_1$. Since $page_1$ is currently accessed only by $GPU_0$, it is considered a private page. Consequently, the OP-Controller only checks the host page table, bypasses the O-Table, and directly resolves the page fault by migrating $page_1$ to $GPU_0$ adopting on-touch migration (①). Then, $GPU_1$ requests the same page $page_1$ (②), and a page fault to the same page is triggered and sent to the host CPU. The host page table reveals that the data resides on another GPU, not the CPU, confirming that the page is shared. This request is then directed to the O-Table (③). Using the Obj_ID in the virtual address, the corresponding entry in the O-Table is located. Since it is the first time O-Table access for this object and the PF Count is initially "000" for this object, the OP-Controller determines the appropriate policy based on the object pattern, updates the policy bit, and increments the PF Count by 1. For example, if it is a write request, the policy is set to "1", which corresponds to access counter-based migration. The UVM driver then employs this policy to resolve the page fault and update the related PTEs. Next, $GPU_3$ requests $page_1$ (④), and the O-Table is checked after host PT lookup since the page is shared. It finds that the PF count is not zero, therefore, the OP-Controller utilizes the recorded policy from the O-Table to resolve the page fault and increment the PF Count by one (⑤). This process is consistently applied to subsequent accesses. Note that, the object pattern as well as the policy decision are determined by the pattern of only a single page. This is justified by Observation 2 and Observation 4 in Section IV-B, where the access patterns of a particular object's pages remain consistent. Therefore, it is unnecessary to track multiple pages within an object to determine the appropriate policy. Analyzing the pattern of just one page is sufficient to accurately decide the behavior of all pages associated with that object. This simple mechanism effectively removes a significant amount of overhead, both in latency and metadata size, in capturing the object access pattern and determining the management policy.

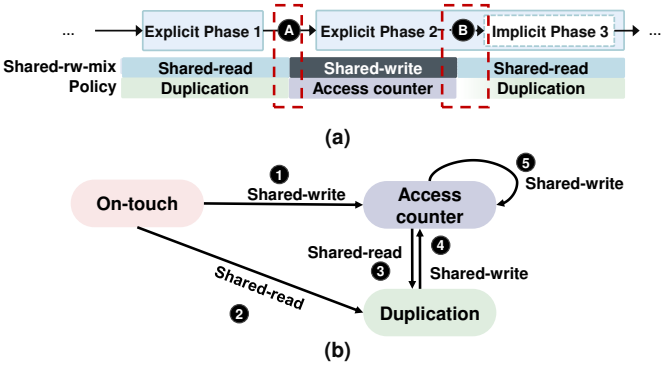



Fig. 13. (a) Example of self-correction. (b) The state transition diagram.

**Self-Correction.** Recall from Observation 4 that the object pattern changes across different phases. To effectively detect changes in object patterns across phases and dynamically apply the appropriate policy, our proposed OP-Controller includes the following two self-correction mechanisms.

For implicit phase detection, we use shared page faults as triggers for resets. A shared page fault is defined as a page fault that occurs to a shared object; only shared faults are propagated to the O-Table. As previously discussed, page faults occurring during the first access to a page do not reach the O-Table. A high frequency of shared page faults for an object suggests that the current policy may no longer

be appropriate due to frequent accesses by multiple GPUs. We use a pre-defined reset threshold to trigger the policy correction[1]. Once the PF Count reaches the reset threshold, the PF Count is reset to "000" and the OP-Controller then re-evaluates the object pattern and determines a more suitable policy based on the new pattern observed. We use Figure 13 (a) as an example to illustrate how the self-correction operates. Assuming a given object is shared-write in an explicit phase (phase-2). The policy employed for the shared-write phase is access counter-based migration. During execution, if the page becomes heavily shared-read by different GPUs (❽), the access counter will increase to the threshold and trigger page migrations and page faults. The page faults will be captured by OASIS, and once the page fault count reaches the reset threshold, the self-correction mechanism detects the implicit phase (phase-3) change, resets the entry in the O-Table, and relearns the policy based on read-access patterns, setting the policy to duplication.

For explicit phase detection, when transitioning to a new phase (e.g., launching a CUDA kernel), the runtime system also resets the corresponding entry in the O-Table (e.g., ❹ in Figure 13 (a)). This integration requires modifying the CUDA runtime API, such as `cudaLaunchKernel`, to include an O-Table entry reset as part of the kernel launch process. Note that, this reset only sets the PF count to "000" which ensures that the new policy is learned at the next page fault. In Figure 13 (a), since the access pattern is changed from shared-read (explicit-phase-1) to shared-write (explicit-phase-2), the policy for this object is re-learned as counter-based migration. Then, the subsequent shared page faults will retrieve metadata from the O-Table and apply the newly learned policy.

Figure 13 (b) depicts the state transition diagram. Initially, an object follows the on-touch policy. If the access pattern remains private, the object remains in the on-touch policy. Once the access pattern changes (i.e., phase changes), depending on whether it is shared-write or shared-read, the policy state will transit to access counter-based migration or duplication, respectively (❶, ❷). Note that, once the policy is changed from on-touch to duplication or access counter-based, it will not return to on-touch. This is because when an object's access pattern changes from shared back to private, it will eventually migrate/duplicate in the requesting GPU, and all subsequent accesses to the object pages will be local without generating any page fault. Reverting to the on-touch policy is therefore unnecessary. For a shared-write object in the access counter-based migration state, if shared write accesses continue, the policy remains unchanged due to its write pattern (❺). However, if the access pattern changes to shared-read, the policy will change to duplication (❸). On the other hand, for a shared-read object using duplication, shared write accesses to this object will trigger page-protection faults, self-correction then reset the policy to access counter-based migration (❹).

[1]Our default setting is 8 faults, we also evaluate our approach with other reset thresholds in Section VI-B1.

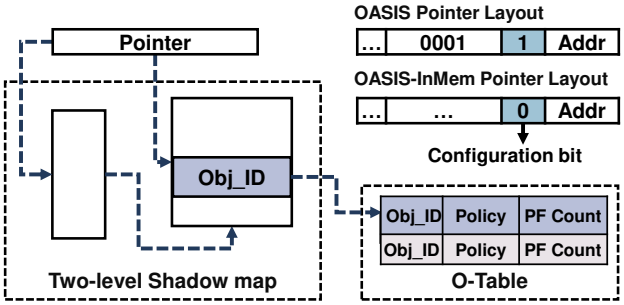



Fig. 14. High level overview of OASIS-InMem.

### E. Overhead

OASIS introduces two main sources of overhead. First, the O-Table storage overhead and associated access latency. Since the number of objects for most of our evaluated workloads is small, we find that a 16-entry O-Table is sufficient. Each entry in the O-Table occupies 12 bits, dedicated to storing object information. Thus OASIS requires an O-Table that requires $12\,\text{bits} \times 16 = 24\,\text{bytes}$. We employ CACTI [52] to estimate chip area, and the results indicate that the O-Table occupies less than 0.02% of the total area compared to a 256 KB 16-way cache. In addition, since the O-Table is implemented as an on-chip hardware component, the latency associated with accessing this table is also negligible. The second overhead is related to updating the PTE policy bits. Updating the policy bits in the PTE of all devices' page tables occurs whenever a shared page fault is directed to the O-Table and a new policy needs to be applied. However, this update process coincides with the page fault resolution activities, which typically involve page migrations, duplications, or collapses. Since these operations already necessitate PTE invalidation or updates, the overhead for updating the policy bits in PTEs effectively runs in parallel with the page fault resolution process. Thus, although OASIS introduces specific overheads, they are either minimal or efficiently integrated into the existing processes, minimizing their impact on the system's overall performance.

### F. OASIS-InMem: A Scalable Software Alternative

While hardware-only OASIS provides fast lookups using the hardware O-Table, it has scalability limitations. When the number of objects exceeds the unused upper pointer bits (i.e., 15 bits) or the upper bits are reserved for other purposes [9], [47], we propose OASIS-InMem, a software-only system that (i) stores the O-Table in system memory (i.e., O-Table-InMem), and (ii) utilizes a shadow map to retrieve the index of the O-Table-InMem entry associated with a given object. Figure 14 shows the details of OASIS-InMem. The shadow map is a software component and is managed by the runtime driver. In OASIS-InMem, we do not encode the Obj_ID in the upper bits of the pointer. Instead, we simply use the configuration bit of "0" to indicate that the Obj_ID is retrieved using the shadow map. By doing so, the upper bits of the pointer remain unused and require no modifications. The shadow map functions similarly to a page table, where an $N$-bit Obj_ID is assigned to each $M$-byte segment of an object's virtual memory region. Here, $M$ is defined by

the unit of memory allocation. To retrieve the Obj_ID for a specific pointer, OASIS-InMem employs a two-level shadow map. The Obj_ID is consistently stored across all shadow map entries that correspond to the memory region allocated to that object. For instance, if we have a shadow map that allocates an $N$-bit Obj_ID for every 4 KB segment of virtual memory, then a 2 MB object would occupy $\frac{2\,MB}{4\,KB} = 512$ shadow map entries, with each entry containing the same $N$-bit Obj_ID. When a pointer accesses memory, OASIS-InMem retrieves the associated metadata by using the pointer value to index into the shadow map and extract the Obj_ID. In brief, OASIS-InMem proposes not only a scalable optimization that accommodates more objects but also a compatible solution that does not use the upper pointer bits.

**Overhead of OASIS-InMem.** OASIS-InMem incurs two main sources of overhead. The first is associated with maintaining the O-Table-InMem in memory. The O-Table-InMem requires $(4 + N) \times \#Obj$ bits of space. For instance, if the number of objects is $\#Obj = 1,024$, the space required amounts to only 1,792 bytes, which is negligible. The second source of overhead originates from the two-level shadow map. The first level of this shadow map is a 128 MB array consisting of $2^{24}$ elements, with each element capable of holding a 64-bit pointer to a second-level shadow map table. The second-level shadow map tables are allocated dynamically and consist of $2^{12}$ N-bit entries where each entry covers 4 KB of allocated virtual memory. Assuming $N = 16$, each second-level entry uses 16 bits that can encode the index of up to $2^{16}$ objects. For a program memory footprint of 64 GB, the number of the corresponding second-level shadow map tables is $\frac{2^{36}}{2^{12} \times 2^{12}} = 2^{12}$, making the total memory usage for the second level equals $2^{12}$ table $\times$ $2^{12}$ entry $\times$ 2 bytes $=$ 32 MB. Summing up the memory usage across both levels, the overall shadow map system incurs a memory overhead of approximately 160 MB (128 MB from the first level plus 32 MB from the second level). This represents less than 0.3% of the total 64 GB allocated memory. Moreover, memory access latency for the shadow table and O-Table-InMem is minimized by caching both in the CPU's last-level cache (LLC). As most program data resides in the GPU, the CPU's LLC is underutilized, making it feasible to efficiently cache these table accesses. This significantly minimizes latency.

## VI. Evaluation

### A. Overall Performance

We evaluate our proposed design using the benchmarks listed in Table II. The baseline configuration is presented in Table I. Figure 15 plots the performance of OASIS and three policies (i.e., on-touch migration, access counter-based migration, and duplication), and the results are normalized to the baseline on-touch migration. OASIS achieves an average of 64%, 35%, and 42% performance improvement compared to uniformly adopting on-touch migration, access counter-based migration, and duplication, respectively. The performance improvement is because OASIS effectively captures the object

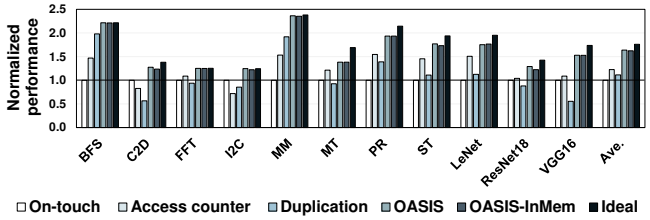



Fig. 15. Performance of different page-management policies versus OASIS relative to baseline on-touch migration.

patterns and their variances across different phases. First, our approach effectively identifies the most suitable page management policies for different objects across various applications. For example, in the `MM` application, the three main objects are MM_A, MM_B, and MM_C. Our approach effectively identifies the patterns of these objects, applying duplication to shared-read-only MM_A and MM_B, and on-touch migration to private-write-only MM_C. OASIS is similarly effective for other applications that consist of objects operating in a single phase without implicit phases, such as `FFT`, `I2C`, `MM`, `MT`, and `PR`. Second, our approach effectively captures object pattern changes during execution and dynamically assigns the most appropriate policies accordingly. For example, in `C2D`, the object pattern changes due to explicit phase transitions, and different phases utilize the same object in varied ways. Our approach effectively resets the previous policy at each phase change and applies a new, suitable policy for the object. In `ST`, where there are no explicit phase changes, OASIS captures the phase transitions through page faults. It recognizes the swapped *currData* in each iteration, applying duplication to the read-only *currData*. Our evaluated benchmarks have a mixed number of explicit and implicit phases detected by OASIS. On average, there are 67 explicit and 11 implicit phases. For example, there are 8 explicit phase changes in `C2D`, and 129 explicit phase changes in `LeNet`. In contrast, `ST` has 20 implicit phases. One can observe that most of these applications reach the ideal performance. However, the performance of some applications still exhibits a gap from the ideal due to the presence of shared-rw-mix and shared-write-only objects. These objects cannot be partitioned into different implicit phases for finer granular pattern recognition. As we discussed in Section IV-B, private and read-only objects can achieve ideal performance, whereas the shared-rw-mix and shared-write-only objects cannot.

We also plot the performance of OASIS-InMem normalized to the baseline on-touch migration in Figure 15. One can observe that the performance of OASIS-InMem drops only 2% on average compared to OASIS, and it still significantly outperforms other policies. The effectiveness of OASIS-InMem is maintained by leveraging the host CPU's LLC to mitigate latencies associated with shadow memory and O-Table-InMem access. However, as the initial accesses to the shadow map and O-Table-InMem are unavoidable, they introduce some overhead, leading to a slight performance degradation. Overall, OASIS-InMem remains highly effective. Note that we employ simulation to validate OASIS-InMem and demonstrate its

feasibility as a proof-of-concept. A comprehensive software implementation on a real system, including optimizations in the UVM drivers and Linux kernel, falls beyond the scope of this paper and is planned for future work.

### B. Sensitivity Analyses

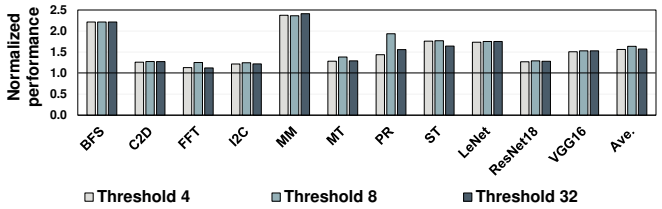



Fig. 16. OASIS using 4, 8, and 32 as the reset threshold.

*1) Different reset threshold:* Recall that we utilize a shared page fault threshold to reset the O-Table entry, enabling the runtime OP-Controller to relearn the most effective page management policies for objects. A higher threshold value means more shared page faults are required to trigger self-correction for O-Table, which may delay the system's response to object patterns due to implicit phase shifts. Conversely, a lower threshold value leads to more frequent resets of the O-Table entries, potentially resulting in unnecessary updates to the policies. To determine the most effective reset threshold, we evaluated system performance using different reset thresholds (4, 8, and 32), normalized to the baseline on-touch migration, as shown in Figure 16. The performance improvements over the baseline were 55%, 64%, and 56% for thresholds of 4, 8, and 32, respectively. We observed that performance gains tend to saturate at a threshold of 8, which we then adopted as the default setting in our main results. Additionally, one can observe that some applications are less affected by the threshold. This is because these applications typically consist of read-only or private objects and without phase changes. Once the policy is set, even if the policy is reset by reaching the fault threshold, it remains the same as the previous optimal policy. However, for some applications, the choice of threshold influences performance. These applications contain shared-rw-mix objects. Since we use only one page to determine the pattern of the entire object, a threshold set too low can cause the policy to fluctuate between duplication and access counter-based migration, introducing unnecessary page-collapse overhead. Conversely, a threshold set too high may delay the adoption of a more suitable policy.

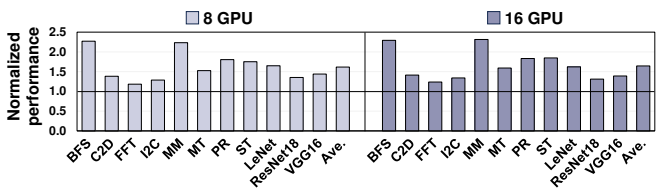



Fig. 17. OASIS with 8 and 16 GPUs.

*2) GPU Count:* We evaluate OASIS in 8-GPU and 16-GPU systems. Figure 17 presents the performance of OASIS using 8 and 16 GPUs, each normalized to their respective baselines. Note that we proportionally increase the workload size to scale up to 8 and 16 GPUs. Table III shows the increased memory footprints for each application. OASIS achieves performance improvements of 65% and 67% over on-touch migration when using 8 and 16 GPUs, respectively. The results indicate our approach remains effective with different numbers of GPUs. This is because even with more GPUs, the object patterns still align with the patterns we discussed in Section IV-B, and our proposed approach effectively captures these patterns, allowing it to assign the most suitable per-object policy.

TABLE III
MEMORY FOOTPRINT (MB) FOR DIFFERENT GPU COUNTS.

| Abbr. | BFS | C2D | FFT | I2C | MM | MT | PR | ST | LeNet | VGG16 | ResNet18 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **8-GPU** | 64 | 200 | 96 | 175 | 128 | 160 | 74 | 65 | 64 | 358 | 508 |
| **16-GPU** | 128 | 308 | 192 | 264 | 192 | 320 | 132 | 129 | 170 | 718 | 1167 |

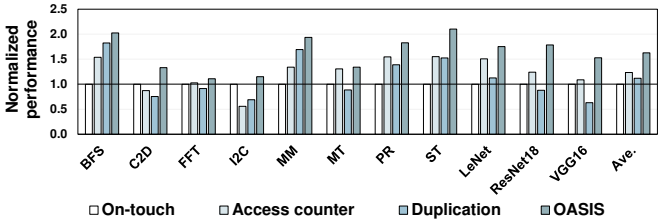



Fig. 18. OASIS with larger input size.

*3) Input size:* We next evaluate OASIS' effectiveness on large input sizes. To do that, we use the 16-GPU input size (Table III) and run the experiments on a 4-GPU system under each page management policy. As shown in Figure 18, OASIS achieves an average 62% performance improvement over the baseline, indicating that OASIS remains effective for larger workloads. This is because large input sizes translate into larger object sizes. Increasing the object size does not necessarily significantly affect object behavior. Consequently, OASIS can efficiently track and record data for those objects, and remain effective for large inputs.

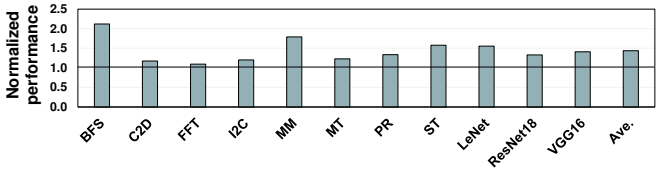



Fig. 19. OASIS with 2 MB pages.

*4) Large pages:* We evaluate OASIS using a 2 MB page. The result is shown in Figure 19. The performance of OASIS using 2 MB pages is normalized to the baseline on-touch migration with 2 MB pages. The average performance improvement is 43% compared to the baseline on-touch migration using 2 MB pages. This indicates OASIS remain effective even when using larger pages. However, one can observe that the performance improvement is reduced. This is because the use of a 2 MB page may convert a private object to a shared one. For instance, consider an object composed of 10 4 KB pages: $GPU_0$ privately reads and writes the first five pages, while $GPU_1$ does the same with the other five. Now this object is a private-rw-mix object. However, assuming these pages are consolidated into a single 2MB page, all ten pages

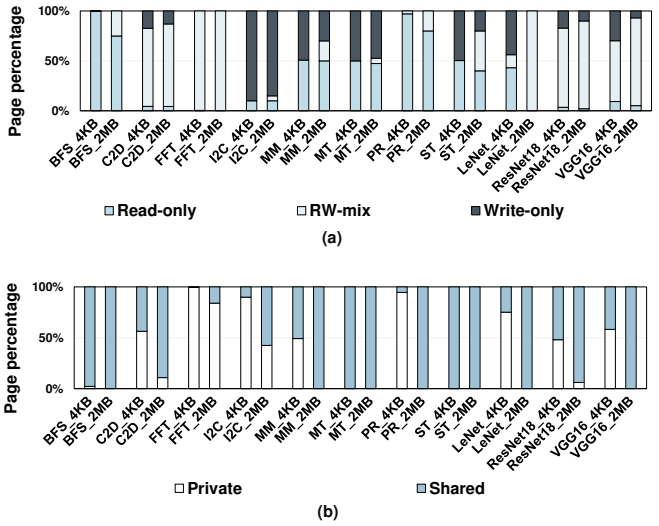



Fig. 20. Percentage of different types of pages.

become shared by $GPU_0$ and $GPU_1$, resulting in a shared-rw-mix pattern for the object. As previously discussed, while on-touch migration achieves ideal performance for private-rw-mix objects, none of the three policies can enable shared-rw-mix objects to reach ideal performance. Consequently, the increase in shared-rw-mix objects further limits the potential for performance improvement. We also characterize the page access patterns with 2 MB pages in Figure 20. Figure 20 (a) shows the read or write percentage of pages, while Figure 20 (b) illustrates the percentage of private versus shared pages. One can observe that the percentage of shared and rw-mix pages is higher with 2 MB pages compared to 4 KB pages.

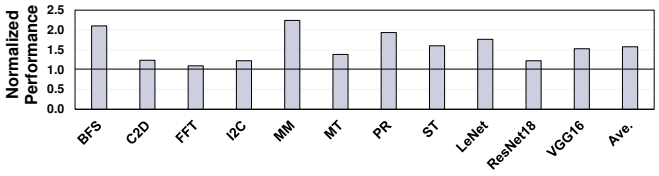



Fig. 21. OASIS with initial page placement on GPU.

*5) Initial page placement:* In our baseline configuration, pages are initially placed on the host. We study sensitivity to initial page placement by distributing pages uniformly across the GPUs (both in baseline and OASIS). As illustrated in Figure 21, OASIS achieves a 57% average performance improvement over the baseline on-touch migration, indicating that OASIS is insensitive to the initial data placement.

### C. Comparison to GRIT

We compare OASIS with the state-of-the-art multi-GPU page management, i.e., GRIT [53]. GRIT has three components: (i) Fault-Aware Initiator to determine when the current page management policy is inappropriate and needs to be changed, (ii) Policy Decision Selection to decide which policy to change to, and (iii) Neighboring-Aware Prediction to proactively decide the management policy of neighboring pages leveraging the spatial locality. Figure 22 compares the performance of OASIS normalized to GRIT. For fair

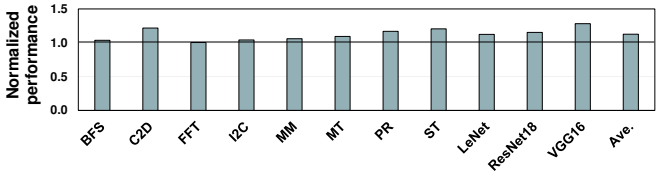



Fig. 22. Comparison to GRIT [53].

comparison, we use the same application input size for these two approaches. OASIS achieves a performance improvement of 12% over GRIT, attributable to the following key factors. First, under the same input size settings, OASIS manages fewer objects relative to the thousands of pages that GRIT handles. This allows OASIS to store object information within a small hardware component, significantly reducing memory access overhead. Second, OASIS utilizes phase changes to trigger the re-determination of page management policies, and efficiently capture the optimal timing for policy adjustments. In contrast, GRIT requires four faults to trigger a policy change for a single page. Although OASIS also uses page faults as indicators for implicit phase changes, it only requires a few page faults per object for a reset, whereas GRIT needs a page fault for each page reset. This makes OASIS more effective in responding to dynamic changes. In addition, OASIS improves effectiveness by reducing on-chip storage overhead, metadata overhead, and design complexity. Specifically, GRIT requires 48 bits (per-page) of in-memory storage to store page attributes while OASIS only requires 12 bits (per-object). Since the number of objects is much fewer than the number of application pages, OASIS has negligible impact on the reduction in effective GPU memory capacity. GRIT requires a 352-byte of on-chip caching structure (i.e., PA-Cache). OASIS on the other hand requires only 24 bytes on-chip (i.e., O-Table). Note that OASIS-InMem eliminates all on-chip storage overhead. Moreover, OASIS applies the same policy to all pages of a given object, thereby effectively avoiding the design complexity that GRIT incurs for predicting the neighboring page policy. Overall, OASIS proves to be more effective than page-granularity analysis in terms of space efficiency, time effectiveness, and design complexity.

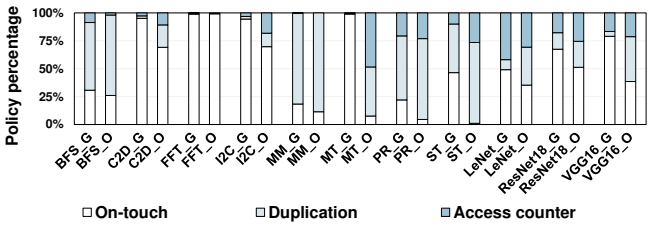



Fig. 23. Page policy distribution under GRIT and OASIS.

We present a breakdown of the page management policies applied to all L2 TLB miss requests for OASIS and GRIT, as illustrated in Figure 23. To better illustrate the performance improvement, we also present a comparison of the total number of GPU page faults for both OASIS and GRIT in Figure 24. The number of page faults plays a crucial role in performance, as each fault incurs a significant overhead due to CPU interruptions and UVM handling. Reducing page faults

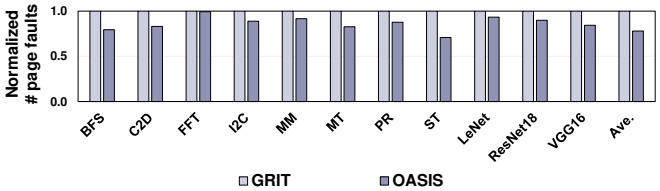



Fig. 24. The number of page faults.

can lead to performance gains. As shown, OASIS achieves a reduction of 22% in page faults compared to GRIT, which contributes to its overall improved performance.

### D. Oversubscription

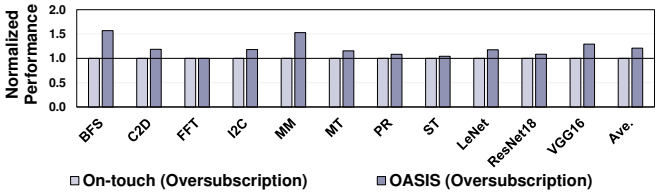



Fig. 25. Performance of baseline and OASIS under oversubscription.

Recall that OASIS relies on the physical address range to distinguish between private and shared pages. However, when a shared page is evicted to CPU memory under oversubscription, it is incorrectly identified as private when re-referenced and leads to inaccurate policy and re-learning overheads. To resolve this issue, we also leverage policy bits in the PTE to manage pages during oversubscription. If the policy of pages on the host CPU differs from on-touch migration (which is the default initial policy), they are treated as shared pages, and the O-Table is used to determine the appropriate policy for this object. To evaluate OASIS under oversubscription without incurring excessively long simulation times with large memory footprints, we keep the working set of the applications unchanged. Instead, we configure a parameter that specifies the available free space in the GPU memory, simulating a scenario with 150% memory oversubscription [11], [22], [23]. As shown in Figure 25, OASIS achieves a 20% performance improvement over the baseline on-touch migration when oversubscription occurs, demonstrating that OASIS can still deliver improved performance under memory oversubscription. It is worth noting that memory oversubscription is very expensive and can dominate the execution overheads, offsetting the performance gains delivered by OASIS.

## VII. Related Work

**Object-Based Optimizations.** Prior research has effectively leveraged object-level information to optimize CPU and GPU systems across different contexts [14], [28], [31], [45], [50]. For example, several studies have maintained object metadata in tables and utilized this information to enforce memory safety on GPUs [31], [50] and CPUs [28], [45]. Additionally, object-based information has been used for reducing the TLB translation cost of contiguous physical memory allocations [14]. To the best of our knowledge, OASIS is the first work to leverage object-level information for enhancing page management decisions in multi-GPU systems.

**Multi-GPUs and Optimizations.** Multi-GPUs are already widely employed to enhance GPU performance across a broad spectrum of workloads [29], [34], [46]. Substantial prior studies have explored approaches to improve the runtime memory management in multi-GPU systems [1], [2], [56]. Young et al. [56] improved NUMA-GPU performance of multi-GPU systems with CARVE (caching remote data in video memory), a technique that allocates a small portion of GPU memory to store contents from remote memory. However, none of these studies focus on runtime object patterns, which are crucial for further improving runtime memory management. In our work, we provide a comprehensive analysis of object memory access patterns and leverage these insights to optimize page management across multi-GPU systems.

**Static Analysis.** Static analysis and `cudaMemAdvise` allow the system to make informed decisions about data placement and migration, optimizing memory access patterns and reducing overhead in heterogeneous systems [12], [43]. These techniques can be employed to determine whether an object is read or written to, thus providing valuable hints for efficient memory management. However, neither static analysis nor `cudaMemAdvise` can determine whether an object is private or shared at runtime. Making this distinction requires runtime support, which is an essential component of our approach in deciding between the different policies.

## VIII. Conclusion

In this paper, we conduct a comprehensive study of memory access patterns at the data object granularity on multi-GPU systems. Our investigation reveals that the diverse object patterns, along with variations in these patterns across applications and within the execution of a single application, significantly affect the performance of different page management policies. We propose OASIS, which dynamically tracks and manages object patterns, automatically adjusting page management policies for objects. Experimental results demonstrate that OASIS significantly improves performance, delivering average improvements of 64%, 35%, and 42% over uniformly employing on-touch migration, access counter-based migration, and duplication, respectively. Moreover, OASIS outperforms state-of-the-art GRIT by 12% and delivers near-optimal performance.

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

## 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 607–618.

[3] 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.

[4] 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 (SC)*, 2021, pp. 1–15.

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

[6] AMD. (2024) AMD64 Architecture Programmer's Manual Volume 2: System Programming. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf

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

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

[9] ARM, "Memory tagging extension: Enhancing memory safety through architecture," https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety, 2019.

[10] ARM. (2024) Arm Architecture Reference Manual for A-profile architecture. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ka

[11] 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 596–609.

[12] T. Brokhman, P. Lifshits, and M. Silberstein, "GAIA: An OS page cache for heterogeneous systems," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 661–674.

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

[14] D. Chen, D. Tong, C. Yang, J. Yi, and X. Cheng, "Flexpointer: Fast address translation based on range TLB and tagged pointers," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 2, March 2023. [Online]. Available: https://doi.org/10.1145/3579854

[15] Y. Dai, X. Tang, and Y. Zhang, "FlexGM: An adaptive runtime system to accelerate graph matching networks on GPUs," in *IEEE International Conference on Computer Design (ICCD)*, 2023, pp. 348–356.

[16] 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*, Pittsburgh, Pennsylvania, USA, 2010, p. 63–74.

[17] 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.

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

[19] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the GPU — past, present and future," *Medical image analysis*, vol. 17, no. 8, pp. 1073–1094, 2013.

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

[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 *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1212–1217.

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

[24] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of GPUs and multicores," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: ACM, 2014, pp. 413–423.

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

[26] Intel. (2018) The Future of Core, Intel GPUs, 10nm, and Hybrid x86. [Online]. Available: https://www.anandtech.com/show/13699/intel-architecture-day-2018-core-future-hybrid-x86/5

[27] Intel. (2022) Architecture Instruction Set Extensions and Future Features Programming Reference. [Online]. Available: https://cdrdv2-public.intel.com/671368/architecture-instruction-set-extensions-programming-reference.pdf

[28] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1153–1166.

[29] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[31] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim, "Securing GPU via region-based bounds checking," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 27–41.

[32] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 1163–1177.

[33] S. Li, C. Wu, A. Li, Y. Wang, X. Tang, and G. Yuan, "Waxing-and-waning: A generic similarity-based framework for efficient self-supervised learning," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=TilcG5C8bN

[34] L.L.N. Laboratory, "Coral Benchmarks," https://asc.llnl.gov/CORAL-benchmarks/, 2014.

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

[36] 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.

[37] 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

[38] 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#

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

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

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

[42] 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

[43] NVIDIA Corp. (2024) Cuda runtime API documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

[44] OSDev Community. (n.d.) Exceptions. [Online]. Available: https://wiki.osdev.org/Exceptions

[45] G. Saileshwar, R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu, "HeapCheck: Low-cost hardware support for memory safety," *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 1, jan 2022.

[46] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[47] M. B. Sullivan, M. Tarek Ibn Ziad, A. Jaleel, and S. W. Keckler, "Implicit memory tagging: No-overhead memory safety using alias-free tagged ECC," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, June 2023.

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

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

[50] M. Tarek Ibn Ziad, S. Damani, A. Jaleel, S. W. Keckler, and M. Stephenson, "cuCatch: A debugging tool for efficiently catching memory safety violations in CUDA applications," *Proceedings of the ACM on Programming Languages, Issue. PLDI*, vol. 7, no. 111, June 2023.

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

[52] 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 *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2008, pp. 51–62.

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

[54] Ya Le and Xuan Yang. (2015) Tiny imagenet visual recognition challenge. [Online]. Available: http://cs231n.stanford.edu/

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

[56] 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 *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 339–351.

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