Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems

Trinayan Baruah[1] Yifan Sun[1] Ali Tolga Dinçer[3] Saiful A. Mojumder[2],
José L. Abellán[4] Yash Ukidave[5] Ajay Joshi[2] Norman Rubin[1] John Kim[6] David Kaeli[1]
[1]Northeastern University [2]Boston University [3]Istanbul Technical University,
[4]Universidad Católica San Antonio de Murcia, [5]Millennium USA, [6]KAIST
[1]{tbaruah, yifansun, kaeli}@ece.neu.edu, nrubin3@gmail.com [2]{msam, joshi}@bu.edu
[3]dincer15@itu.edu.tr [4]jlabellan@ucam.edu [5]yash.ukidave@mlp.com [6]jjk12@kaist.edu

***Abstract*—As transistor scaling becomes increasingly more difficult to achieve, scaling the core count on a single GPU chip has also become extremely challenging. As the volume of data to process in today's increasingly parallel workloads continues to grow unbounded, we need to find scalable solutions that can keep up with this increasing demand. To meet the need of modern-day parallel applications, multi-GPU systems offer a promising path to deliver high performance and large memory capacity. However, multi-GPU systems suffer from performance issues associated with GPU-to-GPU communication and data sharing, which severely impact the benefits of multi-GPU systems. Programming multi-GPU systems has been made considerably simpler with the advent of Unified Memory which enables runtime migration of pages to the GPU on demand.**

Current multi-GPU systems rely on a first-touch Demand Paging scheme, where memory pages are migrated from the CPU to the GPU on the first GPU access to a page. The data sharing nature of GPU applications makes deploying an efficient programmer-transparent mechanism for inter-GPU page migration challenging. Therefore following the initial CPU-to-GPU page migration, the page is pinned on that GPU. Future accesses to this page from other GPUs happen at a cache-line granularity – pages are not transferred between GPUs without significant programmer intervention.

We observe that this mechanism suffers from two major drawbacks: 1) imbalance in the page distribution across multiple GPUs, and 2) inability to move the page to the GPU that uses it most frequently. Both of these problems lead to load imbalance across GPUs, degrading the performance of the multi-GPU system.

To address these problems, we propose *Griffin*, a holistic hardware-software solution to improve the performance of NUMA multi-GPU systems. Griffin introduces programmer-transparent modifications to both the IOMMU and GPU architecture, supporting efficient runtime page migration based on locality information. In particular, Griffin employs a novel mechanism to detect and move pages at runtime between GPUs, increasing the frequency of resolving accesses locally, which in turn improves the performance. To ensure better load balancing across GPUs, Griffin employs a Delayed First-Touch Migration policy that ensures pages are evenly distributed across multiple GPUs. Our results on a diverse set of multi-GPU workloads show that Griffin can achieve up to a 2.9× speedup on a multi-GPU system, while incurring low implementation overhead.

I. INTRODUCTION

GPUs are powerful compute engines for processing data-parallel tasks such as signal processing, large scale simulation, and Deep Neural Networks (DNNs). Over the years, the explosion of Big Data has been able to exhaust the massive compute resources of a single GPU. Integrating more and more transistors on a single die to deliver a larger GPU is becoming extremely difficult [1]. Designing high-performance systems supporting multiple GPUs has been shown to be a promising path forward to further improve application performance [1], [2]. As a result, both industry and academia are looking for better multi-GPU solutions. For example, NVIDIA ships DGX-1 [3] and DGX-2 [4] systems by integrating up to 16 GPUs in each node, targeted mainly at DNN workloads. Similarly, AMD integrates four MI25 GPUs in its TS4 servers [5] to accelerate deep learning applications.

Due to the programming complexity when dealing with multi-GPU platforms, major GPU vendors, such as NVIDIA and AMD have added features to GPU programming frameworks in order to simplify multi-GPU programming. These features include: Unified Memory (UM), System-Level Atomics, and GPU-to-GPU Remote Direct Memory Access (RDMA) [6]. Multi-GPU programming is quickly gaining momentum as newer and simpler multi-GPU programming interfaces that improve system resource utilization are developed [7], [8].

Among the various multi-GPU programming features, UM significantly simplifies multi-GPU programming. UM allows programmers to allocate memory without specifying data placement, and the allocated memory can be used on any CPU or GPU in the system. UM also enables memory oversubscription [9]. Backed by system memory, a programmer can allocate memory exceeding a single GPU's physical memory space, allowing applications that require a large amount of GPU memory to run with existing hardware. With the ever-increasing popularity of UM, improving the performance of UM for multi-GPU systems is critical.

Since UM does not require the developer to move data

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00055

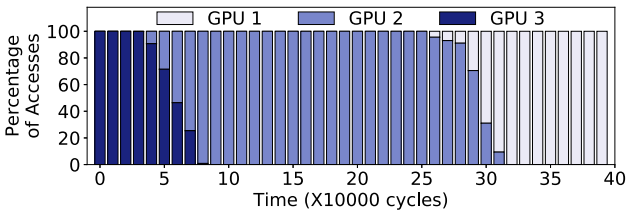

Figure 1: The distribution of the accesses to a page in the Simple Convolution benchmark from multiple GPUs.

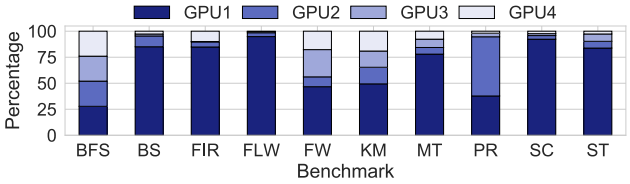

Figure 2: Percentage of pages that get placed in each GPU under first touch policy in a multi-GPU system with four GPUs across different workloads.

explicitly using APIs, the hardware has the responsibility to move the data during execution. Current multi-GPU systems rely on two mechanisms for remote data access: 1) first-touch-based Demand Paging (DP) and 2) Direct Cache Access (DCA). When an application starts, all pages (a page is 4KB-2MB of contiguous memory space) are allocated in the CPU memory. When the first GPU accesses a page, the GPU triggers a GPU page fault and the GPU driver migrates the page to the GPU memory. The page will remain pinned in the GPU until the CPU needs to access it again. If another GPU needs to access the same page, the page is not migrated, but instead, the GPU uses DCA to request the data remotely at a single cache-line granularity (further details in Section II-B).

The motivation behind pinning a page on a GPU after first-touch migration is based on the inherent data-sharing nature of GPU applications, which generate repeated page migrations, significantly impacting performance [10]. Other mechanisms to perform inter-GPU page migration are not completely programmer transparent and require the use of API calls (e.g., cudaMemAdvise [6]), increasing programmer effort.

In summary, the CPU-to-GPU data movement mainly utilizes DP, while the GPU-to-GPU data movement mainly utilizes DCA in current multi-GPU systems [10], [1]. DP and DCA have their unique advantages and disadvantages and complement each other well. DP enables better utilization of data locality, as all the following memory accesses after the first migration are local (i.e., on the same GPU). DP transfers a large amount of data (usually 4KB-2MB) at once over a slow inter-device fabric, introducing long latency. Moreover, DP requires modification to the page table, resulting in instruction pipeline flushes and TLB shootdowns on the device that currently holds the page [11]. TLB shootdown is the mechanism by which the OS ensures that page tables are coherent. Alternatively, DCA does not have to pay the cost of a page migration. However, it does incur the cost of remote memory access for every access to a page residing on another GPU.

In this paper we make two key observations:

1) Existing NUMA multi-GPU mechanisms used to migrate a page only once to a GPU [10], [1], [2] and then pin it on that GPU are inefficient, and could potentially benefit from exploiting runtime locality information to improve performance. As we can see from Figure 1, the number of accesses to a page from a particular GPU can change over time (more details in Section II). Disabling GPU-to-GPU page migration forces other GPUs in the system to access remote data using DCA, which in turn can lead to overall performance degradation. Therefore, enabling GPU-to-GPU page migration can increase the opportunity for GPUs to access pages locally. Although GPU-to-GPU page migration is attractive in terms of reducing the number of remote data accesses, deploying page migration on a GPU is challenging given the high cost associated with migrating a page [11], [10].
2) Our second key observation is that current programmer transparent methods of migrating pages to GPUs using a first-touch policy [2], [10] can lead to severe load imbalance across GPUs, as shown in Figure 2. This imbalance can lead to conflicts and congestion in the inter-GPU fabric since the GPU that holds more of the pages will have to serve more requests from the other GPUs in the system.

The problems above demand a system-wide solution to improve multi-GPU NUMA access with the help of GPU-to-GPU page migration that does not require programmer intervention. Therefore, we propose *Griffin*, a hardware-software solution that improves the GPU microarchitecture, the Input/ Output Memory Management Unit (IOMMU) design, as well as the GPU driver. Griffin's design includes four complementary mechanisms that work in unison to solve these problems in NUMA-based multi-GPU systems. ① To enable efficient inter-GPU migration, we propose a novel configurable page classification algorithm that can categorize and select the best pages for inter-GPU migration. ② To reduce the cost of inter-GPU and CPU-GPU migration, we propose a page migration scheduling policy that can ensure that page migration can be performed with low overhead. ③ In addition, we design and propose a novel GPU pipeline draining mechanism that allows runtime inter-GPU page migration to take place without having to deal with the overhead of flushing the GPU pipeline. ④ Finally, we propose a Delayed First-Touch Migration to ensure that

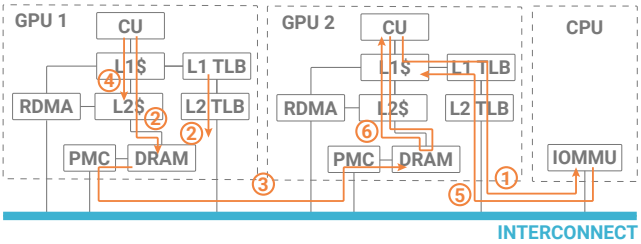

Figure 3: Accessing remote GPU memory with Demand Paging (DP). See Section II for the details of the GPU-to-GPU DP process.

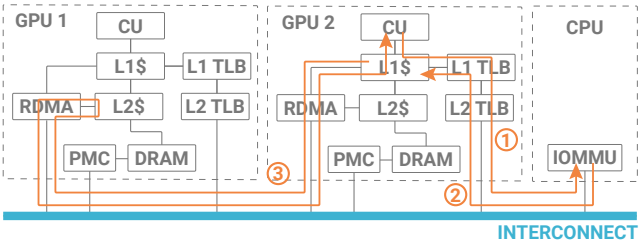

Figure 4: Accessing remote GPU memory with Direct Cache Access (DCA). See Section II for the details of the DCA process.

pages are evenly distributed across GPUs without having to redistribute them after the initial migration from the CPU.

The main contributions of this work include:

- We characterize the key inefficiencies present in current programmer transparent page migration mechanisms designed for multi-GPU systems.
- We present *Griffin*, a holistic programmer-transparent hardware-software solution that can improve the performance of multi-GPU systems. Griffin achieves this by intelligently guiding page placement across multiple GPUs, reducing page migration overhead and evenly distributing pages across the GPUs in the system. To the best of our knowledge, we are the first to propose an efficient inter-GPU page migration mechanism that is completely programmer transparent.
- We implement both the existing NUMA multi-GPU page migration approach and Griffin in MGPUSim [12]. Our evaluation shows that Griffin can improve the overall multi-GPU system performance by up to 2.9×, with very low hardware overhead.

II. Background & Motivation

In this section, we review how existing state-of-the-art multi-GPU systems work and discuss why existing solutions are inefficient for a multi-GPU environment. To demonstrate the inefficiency, we present results from experiments to help motivate our work. Our experimental setup for collecting this data is discussed in Section IV.

A. Multi-GPU Systems

With the slowdown of transistor scaling and the proliferation of big data workloads, single GPU systems can no longer easily achieve performance scaling. Interconnecting multiple discrete GPUs with high-bandwidth interconnects or designing a Multi-Chip-Module (MCM) GPU [1] are promising approaches to harness the power of more computing resources. However, the inter-GPU communication fabric (e.g., PCIe or NVLink) is a major performance bottleneck for NUMA multi-GPU systems. NVLink [13], which is one of the fastest inter-GPU fabrics available on the market, can only provide a maximum bandwidth of up to 300GB/s, which is dwarfed by the local memory bandwidth (1TB/s on an AMD Instinct MI60 using HBM2 [14]). Also, previous work [15] has shown that, in many scenarios, it is difficult to tap into the full bandwidth of the inter-GPU interconnect. Therefore, limiting inter-GPU traffic and reducing remote memory access latency is key to delivering a scalable multi-GPU system. Relying on programmers to optimize programs adds too much burden and is also error prone. Instead, hardware should provide mechanisms that can both simplify multi-GPU programming and improve multi-GPU system performance.

In this paper, we use a Unified Multi-GPU model which allows a GPU program that was originally written for a single-GPU platform to run transparently on a multi-GPU platform. A similar architecture model (i.e., Unified Multi-GPU) has been used in prior studies [7], [8], [1], [2], [10], [12]. Using OpenCL terminology, and assuming a unified multi-GPU memory model, when the user program launches a GPU program (i.e., a kernel), a centralized dispatcher converts the kernel to a 1-D, 2-D, or 3-D grid of threads (work-items). A group of work-items that always execute the same instruction at the same time form a *wavefront*. Wavefronts are executed in Compute Units (CUs), which are hardware components that can run wavefront instructions. A certain number of wavefronts can further be grouped into *workgroups*. Wavefronts from the same workgroup always dispatch work to the same CU. We follow a workgroup scheduling policy, similar to the NUMA GPU systems proposed in prior work [1], [10], [2].

B. Multi-GPU Communication Mechanisms

Communication in multi-GPU systems occurs through two major mechanisms: 1) Page Migration and 2) Direct Cache Access (DCA) through RDMA. These two modes of communication can take place between the CPU and GPU, or between multiple GPUs, and work as follows:

Page Migration: The mechanism to support page migration in GPUs [11] is shown in Figure 3. Assuming GPU 2 needs to access the data that is located in GPU 1's memory hierarchy, the GPU-to-GPU page migration process is as follows. ①: The process starts with one of the GPU 2's

CUs generating a memory access request, with the L1 TLB performing address translation. When the address translation request arrives at the IOMMU for servicing, the IOMMU detects that the page is not present on GPU 2 and triggers a page fault to be handled by the GPU driver. ②: The driver, upon receiving the page fault request, flushes GPU 1 to invalidate the page on GPU 1 for ensuring translation coherence. The flushing process includes flushing in-flight instructions in the CU pipeline, in-flight transactions in the caches and TLBs, and the contents of the caches and TLBs. ③: The Page Migration Controller (PMC) migrates the page from GPU 1 to GPU 2 and notifies the driver about the completion of the page migration. ④: Then, the driver can let GPU 1 replay the flushed memory transactions before continuing execution. ⑤: Now that the page table has been updated with the newly migrated page, it can send back the updated address translation to the L1 and L2 TLBs of GPU 2. ⑥: Finally, with the updated translation information, GPU 2 can finish the memory access. We cannot skip the GPU flushing process for three reasons: 1) the TLB may buffer stale translation information, 2) the L2 cache may hold data that is more recent than the data in the DRAM, 3) in-flight memory transactions in the L1 caches, the L2 caches and the TLBs can have stale translations that have to be discarded. A similar model of page migration also applies for page migration between a CPU and GPU [11].

Direct Cache Access through RDMA: In this mode, the requested data is accessed remotely using the interconnect between the devices (i.e., the CPUs and GPUs) without migrating the page. The process primarily involves three steps, as shown in Figure 4. ①: When a GPU's CU needs to access data, a read or write request is sent to the L1 cache. The L1 cache translates the address with the help of the L1 and L2 TLBs. Since the data is not in the current GPU, the translation request is eventually sent to the IOMMU. ②: Rather than triggering a Page Fault, the IOMMU returns the remote GPU physical address. ③: Receiving the translation for a physical address on a remote GPU, the L1 cache redirects the request to the RDMA engine to fetch data from the remote GPU's L2 cache. The data is returned to the requesting GPU. As the translation does not belong to the physical address space owned by the requesting GPU, and TLBs are not kept hardware coherent in current GPUs, the translation is not cached.

C. Current Challenges

In this section, we identify and present four major challenges that plague current multi-GPU system performance, preventing these systems from achieving good scaling. They are as follows:

1. Non-programmer transparent inter-GPU page migration: Due to the nature of data sharing among GPU applications, a simple page migration scheme based on demand paging alone is inefficient in multi-GPU systems [10]. Previous work on multi-GPU systems [10], [2], [1] has advocated for mechanisms that pin pages on the GPU after an initial migration based on first touch information. Future accesses to this page from other GPUs use Direct Cache Access [10] as described above. A major problem with pinning the page location based on a first-touch policy is that the page cannot be dynamically moved to another GPU based on the dynamic access patterns. Enabling efficient inter-GPU page migration requires programmer intervention and is time-consuming [6].

To illustrate how a page is moved between GPUs, we analyze how a particular page in the Simple Convolution (SC) benchmark [12] is accessed. SC is a representative machine learning workload that can be executed on multi-GPU systems. As shown in Figure 1, the distribution of accesses to the same page by different GPUs can vary over time. The first-touch policy migrates the page first to GPU 3. At the beginning of the application, most of the accesses to this page are from GPU 3 only. However, as time progresses, the number of accesses to this page from GPU 3 becomes negligible, whereas the number of accesses from GPU 2 start to increase. After some time, the number of accesses from GPU 2 starts decreasing, and then the page is mostly accessed by GPU 1. However, the page still continues to reside in GPU 3's memory, requiring GPU 1 to perform a large number of remote accesses for that page. This example helps illustrate the current inefficiencies of the first-touch policy, where pages cannot migrate between GPUs in a programmer transparent fashion. A more intelligent page migration scheme should be able to detect such changes in the access pattern to a page, and then migrate the page transparently to the GPU that accesses it the most. Allowing the page to migrate between GPUs will provide more opportunities for the GPUs to exploit temporal locality, thereby improving performance.

2. First-Touch Demand Paging: By default, Unified Memory (UM) first allocates memory on the CPU, allowing the CPU to initialize the data. When any GPU attempts to access the data, the address translation request will be sent to the IOMMU and will trigger a Page Fault to be handled by the GPU driver. The GPU driver waits until the CPU is not actively accessing the page and then migrates the page to the GPU. Since the first GPU that attempts to access the page acquires the page, the policy is called a *first-touch policy*. The first-touch policy can lead to an imbalance in page assignment. In our experiments that count the percentage of the pages that are migrated to each GPU using the first-touch policy (see Figure 2), we find patterns in many applications where first-touch policy *favors* a certain GPU and assigns a large number of pages to that GPU. Even when we guarantee that the workgroup scheduler evenly distributes workgroups across all GPUs, a large portion of pages end up being placed in one or more GPUs.

The imbalance in page assignment is mainly caused by

Figure 5: Example status of the page table walkers in an IOMMU with one completed page table walk and three ongoing page table walks.

the slight differences in start times of the wavefronts on each GPU. Even though the dispatcher services workgroups in a round-robin fashion across all the GPUs, GPU 1 always requests the first work-group in each round, acquiring a slight "advantage" in the competition for pages. Besides, the network arbiter can also contribute to the imbalance. The GPU that generates requests the fastest may be more likely to be selected by the network arbiter for servicing, and this in turn, makes the GPU generate requests even faster. As a consequence, an excessive percentage of pages are first allocated on a single GPU. This imbalance in page assignment has substantial performance implications. It can result in an increase in the number of cache conflicts in the Last Level cache (LLC) of the GPU which holds the cache lines corresponding to the majority of the pages. Since the other GPUs will all be requesting data from this GPU, it will increase congestion in the RDMA engine and the inter-GPU fabric, which further exacerbates the problem. On the other hand, the GPUs which holds only a few pages will experience low utilization of their local memory bandwidth, L2 caches, and inter-GPU links.

3. FCFS Handling of Page Faults: The default IOMMU scheduler, which handles requests that miss in the local GPU's TLB, services requests based on a First Come First Serve (FCFS) basis. In the event of a first-touch access on current multi-GPU systems, a completed page table walk can trigger a page migration request. However, this mode of handling first-touch page migration requests is inefficient, leading to unnecessary TLB shootdowns and flushing on the CPU. During inter-GPU migration, the overhead of TLB shootdowns and pipeline flushing will be incurred by the GPU servicing the migration. The overhead is even higher on GPUs and the recovery cost of flushing can be quite high [11]. As an example, Figure 5 shows the state of multiple page-table walkers in the IOMMU which are servicing page walk requests from two different GPUs. All of these requests are first-touch accesses and will be selected for page migration from the CPU memory. The completed page table walk requests are shown in white, whereas the colored boxes are page table walks that have not yet been completed. The default FCFS scheduler will schedule a page migration request immediately for Load A without taking into consideration that the other three loads (Loads B, C and D) may also trigger page migration requests to the CPU. Thus, assuming these page table walks finish at different times, each of these requests will result in a TLB shootdown and pipeline flush on the CPU. However, instead of scheduling the migration request immediately, if we delay the migration and wait for the pending page table walk to complete, it will result in batching of these page migration requests together. This will result in only a single shootdown and flush on the CPU, which can provide huge performance improvements. In a similar manner, during inter-GPU page migration, batching of page migration requests can significantly reduce the high setup and recovery costs associated with a page transfer.

4. High cost of GPU Flushing: Finally, the high cost of flushing the GPU pipeline [11] before an inter-GPU page migration, can lead to extremely high overheads. This is because a large amount of in-flight work that has to be discarded during a pipeline flush. Alternative approaches are needed that can mitigate this overhead, enabling multi-GPU systems to migrate pages without incurring high overhead.

III. Griffin

Griffin provides a holistic hardware-software solution to tackle issues related to NUMA accesses and page migration in modern-day multi-GPU systems. As shown in Figure 6, Griffin is mainly composed of four parts, including: *Delayed First-Touch Migration (DFTM)*, *Cooperative Page-Migration Scheduding (CPMS)*, *Dynamic Page Classification (DPC)*, and *Asynchronous Compute Unit Draining (ACUD)*. Among the proposed four components, DFTM and DPC are targeted at finding the best page to migrate, while CPMS and ACUD are targeted at reducing page migration overhead.

Figure 6 shows how the 4 components of Griffin work together. When a Compute Unit (CU) requests an address translation ①, the IOMMU uses multi-threaded Page Table Walkers (PTW) to find the page in the page table. If the IOMMU detects the page is in the CPU, the page fault is transferred to DFTM ② to check if the page should be migrated or not. If DFTM decides the page needs migration, the migration request is passed to CPMS. Rather than migrating the page immediately, CPMS will wait for the current on-going IOMMU page walking to finish and try to batch page migrations ③.

Alternatively, if the IOMMU detects the page is on another GPU, the IOMMU will always send the remote physical address back to the CU and the CU will then use this translation to access remote memory through RDMA ④. In addition, CPMS defines a page migration period so that the GPU driver (which runs on the CPU) can collect the page access count from the CUs across the different GPUs and, at the end of the period, forward the page access count to the IOMMU ⑤. Once the IOMMU has all the page access count data, the IOMMU incorporates DPC to determine which page should be migrated to which GPU and the decision is sent to the CPMS for finding opportunities to batch migrations ⑥. Finally, CPMS will batch the migration

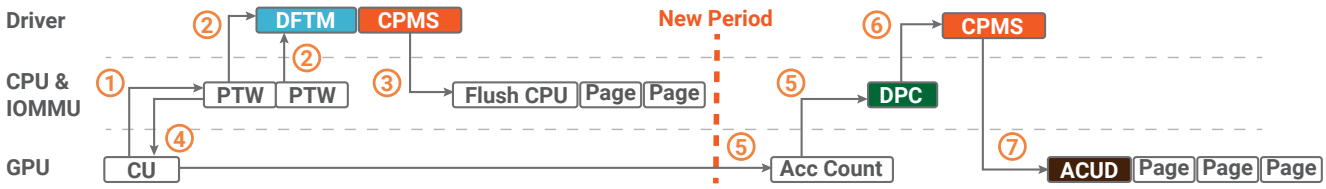

Figure 6: An overview of Griffin showing the four major components of Griffin. The GPU driver runs on the CPU. DFTM handles initial page migration from CPU. CPMS schedules and batches page migrations from CPU to GPU as well as GPU to GPU. DPC decides which pages to migrate. ACUD handles the TLB shootdown process on the GPU from which the page is about to be migrated.

requests to reduce the overhead of page migration. CPMS will then coordinate with the GPU driver to drain the GPU using ACUD ⑦, a special GPU draining approach that is specially designed for page migration, to reduce draining overhead. Overall, the goal is to ensure migration of pages across GPUs at runtime and to do it at low cost.

A. *Delayed First-Touch Migration*

Delayed First-Touch Migration (DFTM) tackles the problem of page assignment imbalance across multiple GPUs, as described in Section 2.3. With DFTM, when a page fault is detected by the IOMMU and the page is resident on the CPU, it will first check the occupancy of the GPU that requests the page. We define occupancy as the ratio of pages in the requesting GPU as compared to the total pages in all the GPUs. If the requesting GPU has the highest occupancy among all the GPUs, the page is not migrated to the requesting GPU. Instead, the IOMMU returns the physical address on the CPU and guides the GPU to access the data using DCA. If the GPU tries to access the page for a second time, the IOMMU triggers a page fault so that the page can be migrated.

By delaying the migration on the first touch, we balance the accuracy of page-migration and node balancing. Denying the GPU that has the highest occupancy restricts the GPU from acquiring too many pages. If a request for this page arrives from a GPU that has low occupancy, then the page is migrated to that low-occupancy GPU, thereby balancing the effective occupancy. The decision to migrate on the second touch rather than waiting longer reduces the number of repeated accesses to CPU memory from the GPU side and reduce long delays waiting for transactions to complete.

B. *Cooperative Page Migration Scheduling*

One of the major costs of page migration is the high overhead of the operations performed before the page is migrated. This cost involves events such as TLB shootdowns, instruction pipeline draining/flushing, and cache flushing. Although runtime page migration can help reduce the remote data transfer, the overheads of page migration can quickly negate that benefit. To reduce this overhead as much as possible, Griffin employs a Cooperative Page Migration Scheduling (CPMS) scheme that coordinates the flushing and page transfers. The main idea of CPMS is to batch page migration requests so that one instance of flush at the source of the pages can be followed by many page migrations.

We batch page migrations in the following two cases:

1. CPU-GPU Page Migration: It is not uncommon for one GPU to generate a stream of page fault requests for the CPU to handle, especially at the beginning of an application's execution phase. The multithreaded nature of the IOMMU's Page Table Walker (PTW) provides ample opportunity for batching [16].

To utilize the batching opportunity presented in CPU-GPU migration, CPMS will not schedule page migration requests immediately when a page fault is generated. Instead, CPMS waits for a certain number of page walks (represented by hyperparameter N_{PTW} in Table I) across the multiple page table walkers to complete. N_{PTW} is set to eight in our simulations since current IOMMU consists of eight page table walkers [16]. Once the page walks complete, CPMS collects all the page faults and schedules CPU flushes and page data transfers by coordinating with the operating system.

2. GPU-GPU Migration: Since the number of candidate pages for migration is usually higher than the number of GPUs, multiple pages are likely to be migrated from one GPU. When one GPU is the common source of multiple page migrations, grouping the migrations can reduce the overhead by incurring the setup cost for page migration at the source GPU only once.

To create the opportunity for batching GPU-GPU migrations, CPMS first disables the "on-demand" page migration between GPUs because it can lead to a lot of back and forth page transfers across the multiple GPUs [10]. CPMS will make decisions about which pages to migrate from one GPU to another and which pages should remain in their current place. As CPMS is implemented in the driver, it can easily acquire global information and make overall decisions.

Next, CPMS divides multi-GPU execution into periods (Section III-C). During each period, GPUs only use DCA to access remote data. Dividing the execution into periods and intelligently migrating the pages can effectively avoid page migration because of a single memory transaction, avoid

page ping-ponging [17] and enable grouping of migrations by only draining a GPU once using ACUD.

CPMS makes migration decisions at the beginning of a migration phase. It first relies on the DPC (to be introduced in Section III-C) to find the candidate pages to migrate. After receiving the candidate page for migration, CPMS decides which GPU to flush and which pages to migrate. According to the configured time between migrations, CPMS limits the number of pages to migrate and the number of GPUs to flush. Thanks to the ACUD (to be introduce in Section III-D) and the GPU-GPU DCA mechanisms, page migration and GPU execution can run concurrently without any significant overhead.

C. Dynamic Page Classification

Before migrating any page, Griffin needs to determine if a page needs to be migrated. We employ a Dynamic Page Classification (DPC) approach that can detect and understand the access pattern to a page made by GPUs. Then, for each type of page, we apply different approaches to determine if the page should be migrated. This mechanism is used to guide the inter-GPU migration.

To begin DPC, we collect access counts from the GPUs. Each Shader Engine (SE) (a group of up to 16 Compute Units and the associated L1 caches) is augmented with a page access counter that monitors the number of memory transactions issued by the Shader Engine. It maintains a table that records the number of post-coalescing memory transactions that access each page. Then, at a predefined interval (number of cycles defined by hyperparameter T_{ac}), the count is collected by the GPU driver and the per-GPU access count is transferred to the IOMMU. Considering that a page ID is 36 bits wide for a 4KB page (48b physical address space minus 12b in-page address offset) and we use 8 bits (overflow keeps the saturating counter value at 0xff) to represent the access count, a message that contains the information for 20 pages (more than enough, based on our experiments) only takes 110 bytes, which is smaller than two cache lines. The access counter is reset to 0 after the count is transferred to the GPU driver.

Our hardware implementation of access counters collects access counts at the L1 cache level since our caches are Virtually-Indexed-Physically-Tagged (VIPT). The implementation of these counters can also be moved up to a higher level of the memory hierarchy if L1 caches are Virtually-Indexed-Virtually-Tagged (VIVT). The basic tenet is that the access counter must be changed before the address translation is done. Hence, the counter location depends on the type of caches being used.

However, raw access counts that are directly collected from the GPUs cannot be leveraged directly for page migration. The counts will vary depending upon the application running and therefore using them directly is not practical. A page may not have a high access count because the wavefront that uses the page is not scheduled to run during a time period. This does not necessarily imply that the page is not going to be used by the GPU in the future. At the same time, a burst of accesses to a page from a certain GPU does not necessarily mean that the GPU needs the page in the future. To solve this problem and to get a more reliable classification counts, we introduce an access count filter that is based on a moving average algorithm [18]. This algorithm is implemented in the IOMMU. A hyperparameter α that varies from 0 to 1 to determine how soon the filter forgets the history access count. A larger α indicates the classifier puts more weight on recently collected data and migrates pages more aggressively, while a smaller α migrates pages more conservatively. Assuming for a certain page p, the filtered access count of the page by GPU g at period n is C_n^{pg}, and the newly collected access count from GPU is N^{pg}, we use the following formula to calculate the new filtered accesses count C_n^{pg}.

$$C_{n+1}^{pg} = (1-\alpha) * C_{n-1}^{pg} + \alpha N^{pg}$$

After the IOMMU updates the filtered access count, Griffin runs its page classification algorithm to select eligible candidates for migration. We propose and design a novel page classification approach to ensure that pages are migrated between devices only if the migration is beneficial for the locality. We classify a page into five different categories as listed below:

1) **Mostly Dedicated Page**: A page is classified as Mostly Dedicated if the page is mainly accessed only by one GPU. To be classified as a Mostly Dedicated Page, the access count from this GPU should be significantly higher (with threshold represented by λ_d) than the GPU with the second-highest access count. In this case, a page should be migrated if it is not on the GPU that has the highest access count.
2) **Shared Page**: A page is classified as Shared if the page is accessed by multiple GPUs without significant variation in the number of access counts to this page. We set the hyperparameter λ_s to classify the page as shared if the highest access count is smaller than λ_s times the second highest access count. This page is a candidate for migration if it is currently located on a GPU that has a very low access count. However, if it is already located on a GPU that has only a slight variation in the access count when compared to the GPU that has the highest access count, then this page is not selected for migration as it is not worth the overhead.
3) **Streaming Page**: GPU workloads can also have memory access patterns that are streaming in nature. We classify a page as streaming if the access counts to this page keeps low (access count under λ_t per cycle) significantly after the initial access. This type of page is also not considered for Inter-GPU migration as the

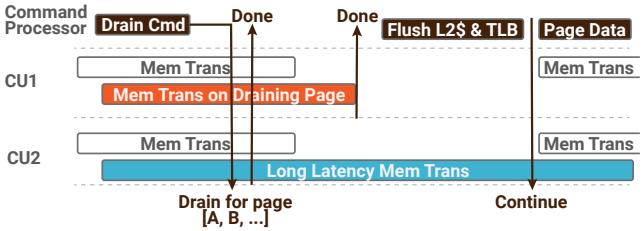

Figure 7: Timeline for Asynchronous Compute Unit Draining.

overhead is too high and there is not enough locality to exploit.

4) **Owner-Shifting Page**: As shown in Figure 1, it is common that a page is previously majorly accessed by one GPU (owner), but as the application progresses, another GPU starts accessing it more often. We classify such pages as "owner-shifting" pages. If the DPC cannot classify the page as the Mostly Dedicated, Shared, or Streaming and the access count of the owner GPU is decreasing while the access count of another GPU is increasing, DPC changes the page class to "owner-shifting". Griffin always migrates the Owner-Shifting pages to the GPU with an increasing access count.
5) **Out-of-interest Page**: When a page cannot be classified by any of the classes above, we are not interested in the page and we do not consider it for migration.

Since the DPC classifies pages and decides which pages to migrate, the candidate pages will be passed to the CPMS component which decides on the best order to drain the GPUs and transfer the data.

D. Asynchronous Compute Unit Draining

One of the major challenges in runtime page migration is the high cost of TLB shootdowns and instruction pipeline flushes. These operations are needed to ensure correctness of the application. Pipeline draining refers to the mechanism where all the in-flight instructions are allowed to complete and the scheduler does not issue any more new instructions to the pipeline. State-of-the-art pipeline draining for GPUs [19], [20] will wait for all pending requests from the current workgroup in the GPU pipeline to complete before marking the drain as completed. However, this is not necessary for page migration. Workgroups can take an extremely long time to complete, depending upon the number of memory transactions they have in flight. Note that some of these requests can end up triggering page migration requests from the CPU, which can take a very long time to resolve. This is one of the reasons why page migration has traditionally relied on pipeline flushing [11] as the page can be migrated immediately after the flush. However, flushing a GPU pipeline drops a large amount of in-flight work on the floor, incurring high recovery costs.

Param	Value	Description
N_{PTW}	8	The number of page walks to wait before triggering page migration.
T_{ac}	1000	Number of cycles between collecting access count.
α	0.03	The rate that the page access count filter forgets the history.
λ_d	2.0	The min ratio between the highest access count and the 2nd highest access count for a page to be considered Mostly Dedicated
λ_s	1.3	The max ratio between the highest access count and 2nd highest access count for a page to be considered Shared
λ_t	0.03	The max number of access per cycle from a GPU for a page to be considered as Streaming

Table I: Default Hyperparameter Configuration.

To avoid the high overhead of pipeline draining and flushing, we propose a low-cost Asynchronous Compute Unit Draining (ACUD). ACUD works as shown in Figure 7. When a page or a group of pages are selected for migration, the GPU driver will send a drain request to the GPU which holds the page. This request carries information about the pages that are about to be migrated.

When a CU receives a drain request, the workgroup scheduler is paused from issuing any new instructions to the GPU pipeline. On a GPU system, every CU maintains a buffer of in-flight memory transactions. The buffer maintains information about the memory addresses that are currently being accessed. These memory addresses are then compared against the memory addresses of the pages that are about to be migrated (contained in the drain request). If there are no pending memory transactions by a CU for the pages that are about to be migrated, the CU will immediately respond with the *Drain Completion* message, without waiting for other memory transactions to finish. Once all the CUs are drained, a TLB shootdown is performed that invalidates the entries for pages which are about to be migrated and flushes the corresponding blocks of data from the L2 cache before initiating the page migration. During L2 flushing, other memory transactions can continue as the flushing affects only the addresses related to the pages that are about to be migrated. The CU drain ensures that there are no existing memory access transactions to any blocks associated with the migrating pages.

ACUD outperforms existing draining/flushing schemes in 4 ways: 1) it allows most of the memory operations to continue during draining, 2) it does not discard any progress from the GPU pipeline that has already been made, 3) compute instructions can overlap with memory transfers, as the *Continue* message is sent before the page data transfer starts, and 4) the ACUD waits the minimum amount of time before the page can be safely transferred.

Component	Configuration	Number per GPU
CU	1.0 GHz	36
L1 Vector Cache	16KB 4-way	36
L1 Inst Cache	32KB 4-way	1 per SE
L1 Scalar Cache	16KB 4-way	1 per SE
L2 Cache	256KB 16-way	8
DRAM	512MB HBM	8
L1 TLB	1 set, 32-way	54
L2 TLB	32 sets, 16-way	1
IOMMU	8 Page Table Walkers	-
Intra-GPU Network	Single-stage XBar	1
Inter-Device Network	32GB/s PCIe-v4	-

Table II: Multi-GPU System Configuration.

IV. Evaluation Methodology

We evaluate Griffin with the multi-GPU simulator MGPUSim (version 1.4.1) [12]. MGPUSim natively supports multi-GPU system simulation and has been validated against AMD multi-GPU systems.

Simulator Extension: We extend MGPUSim to fully support Unified Memory and multi-GPU Demand-Paging. All steps involved in the page migration, such as TLB shootdowns, pipeline flushing/draining, L1/L2 cache flushing, and the page transfer using Page Migration Controller (PMC), have been faithfully modeled in changes to MGPUSim. In our experiments, unless otherwise specified, we use the Griffin hyperparameter values, as listed in Table I.

Evaluating Multi-GPU System: We evaluate a system with 4 AMD Radeon Instinct MI6 GPU [21], with the parameters listed in Table II. Each GPU consists of 4 Shader Engines (SE), with each of them consisting of 9 Compute Units (CUs), making a total of 36 Compute Units per GPU. Each GPU has a multi-level cache hierarchy where L1 caches are private to each CU and L2 cache is shared among the CUs of a GPU. The multiple GPUs in the system are connected with a PCIe-v4 link, providing 32GB/s in each direction.

CUs use virtual addresses, and caches and memory controllers use physical addresses. Address translation is performed with the help of TLBs and the I/O Memory Management Unit (IOMMU). Each CU is equipped with a private L1 TLB and the L2 TLB is shared among all the CUs in each GPU. If the translation misses in the L2 TLB, the translation request is forwarded to the IOMMU (physically located on the CPU) over the PCIe link. The IOMMU consists of multiple page table walkers to service the high memory request rate from the GPUs. We set the page size to 4KB, which is used by most of the current systems as large pages cause higher degree of false sharing as well as page migration overhead [22].

Although MGPUSim does not model CPU execution, we do model the CPU's memory, CPU L2 cache, and the IOMMU on the CPU die. We add a fixed 100 cycle penalty (as used in other studies [11]) for the penalty of flushing the CPU due to page migrations out of the CPU.

For TLB shootdowns on the GPU, we precisely model the penalty. Since MGPUSim models the actual connections between components and communication is performed explicitly through messages, the latency will vary. In reality, the cost of a TLB shootdown on the GPU will also depend on the amount of inflight work that is flushed, as well as the recovery cost incurred every time a TLB shootdown occurs. Our TLB shootdown invalidates only the entries for pages involved in the current migration process as opposed to invalidating the entire TLB. This is because invalidating the entire TLB can lead to a lot of TLB misses, which is extremely expensive on GPUs [23].

Baseline NUMA Multi-GPU System: Our baseline NUMA multi-GPU system follows the mechanism of page migration that has been widely used in previous work [10], [2]. On a first-touch access by a GPU, the page is migrated from the CPU to the GPU. Future accesses to this page does not result in page migrations. If this page is accessed in the future by other GPUs, the communication happens through DCA.

Workloads: We evaluate multiple workloads from AMDAPPSDK [24], Hetero-Mark [25], and SHOC [26] benchmark suite. In addition to the benchmarks that are originally supported by MGPUSim, to explore more irregular memory access patterns, we have extended the simulator to support various graph-based workloads including Floyd-Warshall [24], PageRank [25], and Breadth First Search [24]. The workloads span multiple scientific domains, including machine learning, graph search algorithms, and numerical computations, providing us with a rich corpus of workloads for this study. The workloads also cover a wide range of multi-GPU communication patterns [27].

The memory footprint of our workloads range in size from 30 MB to 64 MB. Extremely long simulation times make it impractical for us to simulate a larger memory footprint. Using 4KB pages on the dataset sizes we are simulating provides thousands of pages in total that are resident across the different devices. This ensures that we can capture the behavior of page migration faithfully in multi-GPU systems.

V. Results

Next, we present the simulation results of the benefits of Griffin as compared to the baseline NUMA multi-GPU systems. We demonstrate how Griffin is able to significantly reduce the overhead of events such as TLB shootdowns, as well as improve the overall occupancy across multiple GPUs. We also show Griffin's dynamic migration scheme in action using an example.

Improvement in Occupancy: As we discussed in Section II, the baseline NUMA-GPU system suffers from imbalance in terms of page distribution on each GPU. In contrast, we can see from Figure 8 that Griffin is able to resolve

Abbv.	Application	Benchmark Suite	Access Pattern	Memory Size
BFS	Breadth First Search	SHOC	Random	32 MB
BS	Bitonic Sort	AMDAPPSDK	Random	36 MB
FIR	Finite Impulse Resp.	Hetero-Mark	Adjacent	64 MB
FLW	Floyd Warshall	AMDAPPSDK	Distributed	44 MB
FW	Fast Walsh Trans.	AMDAPPSDK	Adjacent	40 MB
KM	KMeans Clustering	Hetero-Mark	Partition	51 MB
MT	Matrix Transpose	AMDAPPSDK	Scatter-Gather	44 MB
PR	PageRank Algorithm	Hetero-Mark	Random	38 MB
SC	Simple Convolution	AMDAPPSDK	Adjacent	41 MB
ST	Stencil 2D	SHOC	Adjacent	33 MB

Table III: Workloads used to evaluate the Griffin design.

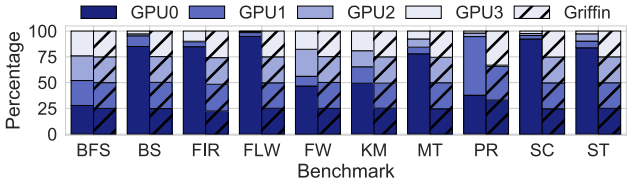

Figure 8: Occupancy balancing improvement. Left hand side bar shows the distribution of pages across four GPUs using the baseline. The right-hand side bar shows the page distribution across four GPUs using Griffin.

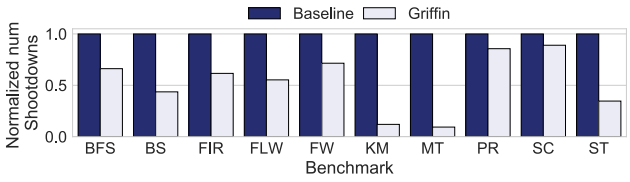

Figure 9: Comparison of the number of TLB shootdowns in the baseline versus Griffin.

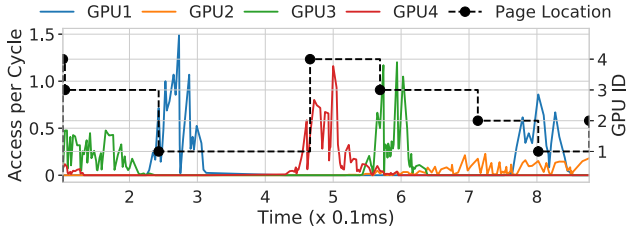

Figure 10: Distribution of accesses to a page in the SC benchmark.

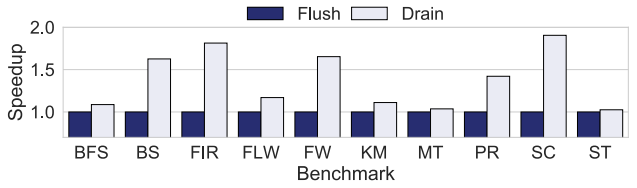

Figure 11: Performance comparison of using Griffin+Flushing vs. Griffin+ACUD.

this imbalance with the DFTM mechanism, achieving a near equal split of pages across all the GPUs without having to do any runtime load balancing.

TLB Shootdowns: TLB shootdowns and pipeline flushing can negatively impact the performance of a system. While Griffin produces additional TLB shootdowns on the GPU due to inter-GPU migrations, the total number of shootdowns is much lower than the baseline NUMA-GPU system, as shown in Figure 9. This shows that the CPMS can effectively batch page migrations and avoid the overhead of costly shootdowns on both the CPU and GPUs.

Dynamic Inter-GPU Migration Decision with DPC: Figure 10 shows Griffin's DPC mechanism in action when running the Simple Convolution benchmark for the most frequently accessed page. The page is primarily accessed by GPU 3 and sparingly by GPU 4. Initially the page migrates from CPU to GPU 4. This is an example that the first-touch mechanism cannot assign the page to GPU3 even though it accesses the page most frequently. Griffin on the other hand migrates the page to GPU 3 in a short time. Later in the execution (≈0.2ms), GPU 3 stops accessing the page and there are more accesses from GPU 1. The black dotted line shows that our proposed mechanism is able to detect this change in page access behavior and respond appropriately by migrating the page to GPU 1. As the access count from GPU 4 start to grow (≈0.46ms), Griffin detects this change and migrates the page to GPU 4. There is a slight delay in the time when the access pattern changes, especially when Griffin migrates a page, as is evident from the black dotted line. This is because Griffin's migration is reactive rather than predictive. A page is not migrated until the DPC recognizes that migration is beneficial. We leave predictive approaches for inter-GPU migration as future work.

ACUD: Figure 11 shows performance difference when using Griffin with our proposed ACUD approach vs a system that employs Griffin with pipeline flushing. The performance difference is quite significant for majority of the benchmarks as the cost of flushing the GPU pipeline can

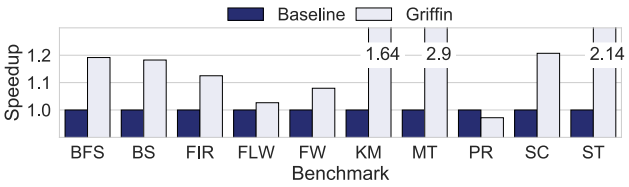

Figure 12: Speedup of Griffin versus the Baseline design.

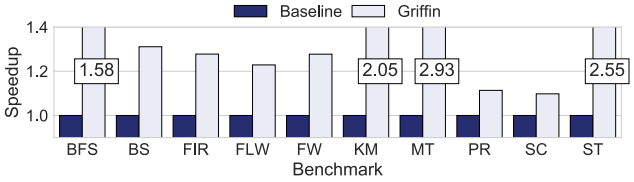

Figure 13: Speedup of Griffin versus the Baseline design with a higher bandwidth interconnect.

be very high. Some benchmarks benefit less from ACUD because sometimes ACUD might still take a long time to drain a GPU. This happens if there are a large number of pages selected for migration and there are ongoing memory transactions on that page from the GPU which is about to be drained. However, it is clear that ACUD always outperforms a system that relies on GPU pipeline flushing for inter-GPU migration.

Overall Speedup: Figure 12 presents the performance benefits of Griffin using the hyperparameters shown in Table I, which we have experimentally determined to be the best set of parameters for our current multi-GPU configuration. We can clearly observe that Griffin significantly outperforms the NUMA-GPU system in a majority of the applications evaluated. Griffin is able to achieve a speedup of 1.37× (geometric mean). Griffin is able to achieve the speedup because of programmer-transparent migration of pages to the GPU by moving the page to the most appropriate GPU and with low cost. The highest speedup is observed for the Matrix-Transpose benchmark, which achieves a 2.9× improvement compared to the baseline. Apart from the benefits of runtime inter-GPU migration, we observe that this application benefits highly from Griffin's DFTM mechanism since most of the pages are not accessed multiple times and Griffin prevents the costly page migrations that lack locality from occurring in the first place. PageRank is the only benchmark where we observe a slowdown when using Griffin as compared to the baseline system. This is mostly because Griffin is not able to migrate the pages to the correct locations. For this application, the access patterns to sparse matrices can be very random and irregular, which makes it difficult to exploit inter-GPU migration effectively.

Some modern multi-GPU systems [3], [4] also employ much higher bandwidth interconnects such as NVLink. These high bandwidth interconnects enable fast remote data access. However, even with such high bandwidth interconnects, having an efficient inter-GPU page migration mechanism is beneficial. From Figure 13 we observe that Griffin still outperforms the baseline design when switching to an interconnect with higher bandwidth. The performance is in fact much better when using a high bandwidth interconnect, especially in benchmarks such as BFS, KM and PR where we observe much improved performance as compared to a system with lower bandwidth. This improvement is due to the fact that Griffin is able to make better use of the higher bandwidth interconnects, which is a result of Griffin's improved page placement.

Hardware Cost: We evaluate the hardware costs of different components of Griffin. Griffin's DFTM requires an extra bit in the page table for each page to mark that it has been accessed once. Once this bit is set, it will remain unchanged until the page is removed from physical memory. The CPMS mechanism of our proposed design does not require any hardware additions, except software-level data structures to orchestrate and batch page migrations among devices. For the DPC mechanism, we require an access count monitor for each Shader Engine (one for every 9 CUs in our design). The access count monitor design basically intercepts memory requests from the CU to the TLB and stores associated metadata. The table consists of 100 metadata entries, with each entry storing the page ID that requires 36 bits, and the 8-bit access count. Each GPU has 4 tables (since there are 4 Shader Engines), resulting in a total storage of 2200 bytes for each GPU. For ACUD, we augment the CU's to enable selective draining when pages are migrated. The CU already keeps a buffer for in-flight memory accesses, since it must keep track of the pending memory requests. We add additional logic for each CU in the form of a 64-bit comparator, as well as arithmetic shifting logic that scans this buffer for pending memory requests and identifies pages under migration. Since the CU does not know the page size being used, the request from the driver provides this information and the CU uses the arithmetic shifting logic to calculate the page size.

VI. Related Work

A. Multi-GPU NUMA Studies

Improving the performance of multi-GPU NUMA systems has recently attracted the attention of many researchers [10], [1], [2], [7], [8]. Prior work [10] has focused on dedicating or allocating extra hardware in the form of caches to address remote access bottlenecks. Scaling such solutions is however challenging. To the best of our knowledge, no previous work on multi-GPU memory systems focuses on solving or improving the performance of demand paging on multi-GPU systems, which is equally important as reducing the remote access bottleneck. Also, in contrast to our work, all of the above work focuses on migrating a page once,

thereby preventing other GPUs in the system to reap the benefits of local accesses to a page. We believe Griffin can also be integrated with previously proposed approaches such as CARVE [10] that focuses on dedicating DRAM space to cache remote data. We leave study of integrated mechanisms for future work. Kim et al. [28] propose CODA, a mechanism which uses a mixture of compiler and runtime techniques to improve the page distribution mechanism across multiple GPUs. Prior work [10] has also considered how to classify pages into two broad categories based on sharing behavior: Read-Shared and Read/Write Shared. However, to the best of our knowledge, we are the first to introduce a novel page classification scheme that is designed to improve runtime decisions for inter-GPU page migration.

B. Runtime Page Migration

Prior work has also studied mechanisms to improve the performance of page migration approaches on CPU systems, as well as emerging heterogeneous memories such as 3D XPoint. Dashti et al. [29] propose Carrefour which is a system that tries to improve the NUMA performance of CPU systems using methods such as interleaving, page replication and page migration using profiling approaches. Agarwal et al. [30] propose Thermostat, a mechanism to migrate pages between heterogeneous memories at runtime based on detecting hot and cold pages. Their approach also relies on page table entry (PTE) invalidations and sampling pages using the Badger Trap tool. Such frequent PTE invalidations can be costly on GPUs, as forcing a TLB miss will stall an entire warp on a GPU leading to low utilization. Dynamic page placement has also been studied by Wilson et al. [31] for CPU-based NUMA systems. Chandra et al. [32] proposed mechanisms to improve the performance for DASH CC-NUMA systems. None of these approaches work well on GPU systems since the cost of page collapses can be very high and runtime software-based page profiling can be expensive on a GPU. Improving page migration performance on a tiered heterogeneous memory system has also been explored [33]. This work focused more on improving transparent migration of huge pages using OS level modifications that can be integrated with Griffin to improve its overall performance. We plan to explore this integration in our future work.

C. Address Translation on GPU

Pichai et al. [34] and Power et al. [35] propose the initial designs to enable virtual memory support on GPUs. Vesley et al. [36] characterize the performance impact of address translation on GPUs, which is non-negligible and can be high for some applications. Previous works [16], [37] also focused on IOMMU enhancements to improve the performance of applications executing on a single GPU. These approaches can be combined with Griffin to achieve better performance. In such approaches the scheduling mechanisms proposed in [16], [37] can work collaboratively with Griffin to handle memory requests issued to the IOMMU from a single GPU as well as multiple GPUs. Jaleel et al. [38] propose a mechanism for increasing the TLB reach for GPUs by caching translations in the GPU LLC and DRAM. Such mechanisms can be integrated with Griffin to reduce the penalty of TLB misses which in turn has the potential to improve performance. Zheng et al [23] propose a novel mechanism to hide the page fault latency on a GPU through CU pipeline modifications and prefetching schemes. Their mechanism can be integrated into Griffin and work alongside the ACUD mechanism. In such systems, ACUD can take care of reducing the cost of pipeline flushes on the GPU where the page is being migrated, whereas their mechanism can help to reduce the penalty of a page fault on the GPU that triggered the page fault request.

D. Methods to reduce TLB shootdowns

Mechanisms to reduce the cost of TLB shootdowns on CPUs, and emerging heterogeneous memory systems, have attracted significant attention over the last decade [39], [40], [41], [42], [43], [44]. This is due to the rising cost of TLB shootdowns, especially as core counts continue to scale and heterogeneous memory makes its way into mainstream systems. Previous work by Agarwal et al. [11] have studied on mechanisms to reduce the occurrence of TLB shootdowns on a CPU-GPU system. Reducing the cost for translation coherence [44] on virtualized systems has also been studied. Romanescu et al. [40] propose a hardware translation coherence protocol for CPU systems In contrast, our work is more concerned with reducing the number of TLB shootdowns on multi-GPU systems, which none of the above prior studies have addressed.

VII. Conclusion

Given the rampant growth of data in emerging applications, multi-GPU servers will become the norm and will be tasked with performance critical applications. As a result, efficient methods to mitigate the NUMA performance bottlenecks of multi-GPU systems will be more and more critical. Programming a multi-GPU system can be challenging given that data placement may not be known apriori. Therefore, improving the performance of unified memory using novel architectural and software solutions is going to be a key enabler to achieve higher performance in multi-GPU systems.

In this work, we have observed that multi-GPU systems can suffer from imbalance across GPUs, as well as the overhead associated with page migrations. We found that enabling intelligent inter-GPU page migration is going to be extremely important for future multi-GPU systems to achieve scalable performance. One of the biggest challenges is how/when to perform pipeline flushes, cache flushes and TLB shootdowns.

Griffin is a hardware-software solution to enable such low cost inter-GPU page migration and resolve imbalances across GPUs. Griffin is comprised of four major components: 1) Delayed First-Touch Migration (DFTM), 2) Cooperative Page Migration Scheduling (CPMS), 3) Dynamic Page Classification (DPC) and 4) Asynchronous Compute Unit Draining (ACUD). These four elements work in harmony to achieve a geometric mean speedup of 1.37× and a peak speedup of 2.9× speedup when compared against current NUMA-based multi-GPU systems. Griffin is able to achieve near perfect load balancing across the multi-GPU system, as well as reduce the cost of TLB shootdowns significantly. Griffin's DPC and CPMS mechanisms enable inter-GPU migration based on runtime profiles to ensure a GPU can satisfy most of its accesses locally. Griffin's ACUD component makes inter-GPU migration more practical, versus current migration mechanisms that rely on pipeline flushing. DFTM ensures that pages are distributed evenly across GPUs and also ensures that pages that are not used more than once are not migrated from the CPU. To further improve upon the performance of Griffin, we plan to consider new components that can predict page accesses by other GPUs and speculatively migrate pages, as well as incorporate page splitting approaches [12], [22].

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback. This work was supported in part by NSF CNS-1525412, NSF CNS-1525474, MINECO TIN2016-78799-P, NRF-2015M3C4A7065647, NRF-2017R1A2B4011457, and AMD.

REFERENCES

[1] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-chip-module GPUs for continued performance scalability," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 320–332, 2017.

[2] 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*, pp. 123–135, ACM, 2017.

[3] N. A. Gawande, J. A. Daily, C. Siegel, N. R. Tallent, and A. Vishnu, "Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing," *Future Generation Computer Systems*, 2018.

[4] NVIDIA, "NVIDIA DGX-2," 2018.

[5] AMD, "AMD Multi GPU box," 2018.

[6] NVIDIA, "NVIDIA Unified Memory," 2018.

[7] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-GPU system design with memory networks," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 484–495, IEEE, 2014.

[8] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 35, 2016.

[9] 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*.

[10] 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)*, pp. 339–351, IEEE, 2018.

[11] 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)*, pp. 354–365, IEEE, 2015.

[12] 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*, ISCA '19, (New York, NY, USA), pp. 197–209, ACM, 2019.

[13] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[14] AMD, "AMD Radeon Instinct MI60 Accelerator," 2018.

[15] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: evaluating modern GPU interconnect via a multi-GPU benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 191–202, IEEE, 2018.

[16] 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 *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 180–192, IEEE Press, 2018.

[17] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "User-level dynamic page migration for multiprogrammed shared-memory multiprocessors," in *Proceedings 2000 International Conference on Parallel Processing*, pp. 95–103, IEEE, 2000.

[18] J. S. Hunter, "The exponentially weighted moving average," *Journal of quality technology*, vol. 18, no. 4, pp. 203–210, 1986.

[19] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 593–606, 2015.

[20] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 193–204, IEEE, 2014.

[21] AMD, "AMD Radeon Instinct MI6 Accelerator," 2018.

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

[23] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for GPUs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 345–357, IEEE, 2016.

[24] AMD, "AMD APP SDK OpenCL Optimization Guide," 2015.

[25] 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)*, pp. 1–10, IEEE, 2016.

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

[27] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, R. Ubal, X. Gong, S. Treadway, Y. Bao, V. Zhao, H. Barclay, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "MGSim+ MGMark: A Framework for Multi-GPU System Research," *arXiv preprint arXiv:1811.02884*, 2018.

[28] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, "CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems,"

[29] 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 SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 381–394, 2013.

[30] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 631–644, ACM, 2017.

[31] K. M. Wilson and B. B. Aglietti, "Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pp. 33–33, ACM, 2001.

[32] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and page migration for multiprocessor compute servers," in *ACM SIGPLAN Notices*, vol. 29, pp. 12–24, ACM, 1994.

[33] 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*, pp. 331–345, ACM, 2019.

[34] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces," in *ACM SIGPLAN Notices*, vol. 49, pp. 743–758, ACM, 2014.

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

[36] 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)*, pp. 161–171, IEEE, 2016.

[37] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular GPU applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 352–363, IEEE, 2018.

[38] A. Jaleel, E. Ebrahimi, and S. Duncan, "Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 1, p. 6, 2019.

[39] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 340–349, IEEE, 2011.

[40] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, IEEE, 2010.

[41] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy translation coherence," in *ACM SIGPLAN Notices*, vol. 53, pp. 651–664, ACM, 2018.

[42] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding TLB shootdowns through self-invalidating TLB entries," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 273–287, IEEE, 2017.

[43] B. Pham, D. Hower, A. Bhattacharjee, and T. Cain, "TLB shootdown mitigation for low-power many-core servers with L1 virtual caches," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 17–20, 2017.

[44] Z. Yan, J. Veselỳ, G. Cox, and A. Bhattacharjee, "Hardware translation coherence for virtualized systems," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 430–443, ACM, 2017.