Appears in the Proceedings of the 2015 International Symposium on High Performance Computer Architecture (HPCA)

Unlocking Bandwidth for GPUs in CC-NUMA Systems

Neha Agarwal[‡] [*] David Nellans[†] Mike O'Connor[†] Stephen W. Keckler[†] Thomas F. Wenisch[‡]
University of Michigan[‡] NVIDIA[†]
nehaag@umich.edu, {dnellans,moconnor,skeckler}@nvidia.com, twenisch@umich.edu

***Abstract*—Historically, GPU-based HPC applications have had a substantial memory bandwidth advantage over CPU-based workloads due to using GDDR rather than DDR memory. However, past GPUs required a restricted programming model where application data was allocated up front and explicitly copied into GPU memory before launching a GPU kernel by the programmer. Recently, GPUs have eased this requirement and now can employ on-demand software page migration between CPU and GPU memory to obviate explicit copying. In the near future, CC-NUMA GPU-CPU systems will appear where software page migration is an optional choice and hardware cache-coherence can also support the GPU accessing CPU memory directly. In this work, we describe the trade-offs and considerations in relying on hardware cache-coherence mechanisms versus using software page migration to optimize the performance of memory-intensive GPU workloads. We show that page migration decisions based on page access frequency alone are a poor solution and that a broader solution using virtual address-based program locality to enable aggressive memory prefetching combined with bandwidth balancing is required to maximize performance. We present a software runtime system requiring minimal hardware support that, on average, outperforms CC-NUMA-based accesses by 1.95×, performs 6% better than the legacy CPU to GPU `memcpy` regime by intelligently using both CPU and GPU memory bandwidth, and comes within 28% of oracular page placement, all while maintaining the relaxed memory semantics of modern GPUs.**

I. Introduction

GPUs have enabled parallel processing for not just graphics applications but for a wide range of HPC installations and data-centers like Amazon's elastic compute cloud (EC2). With this massively parallel processing often comes an insatiable demand for main memory bandwidth as GPUs churn through data at an ever increasing rate. To meet this bandwidth demand, many GPUs have been designed with attached high-bandwidth GDDR memory rather than standard DDR memory used by CPUs. As a result, many GPUs today have GDDR bandwidth that is 2-5× higher than the memory bandwidth available to the CPU in the system. To make best use of the bandwidth available to GPU programs programmers manually copy the data over the relatively slow PCIe bus to the GPU memory, and – only then – launch their GPU kernels. This up-front data allocation and transfer has been necessary since transferring data over the PCIe bus is a high overhead operation, and a bulk transfer of data amortizes this overhead. This data manipulation overhead also results in significant porting challenges when retargeting existing applications to GPUs, particularly for high-level languages that make use of libraries and dynamic memory allocation during application execution.

* A large portion of this work was done when Neha Agarwal was an intern at NVIDIA.

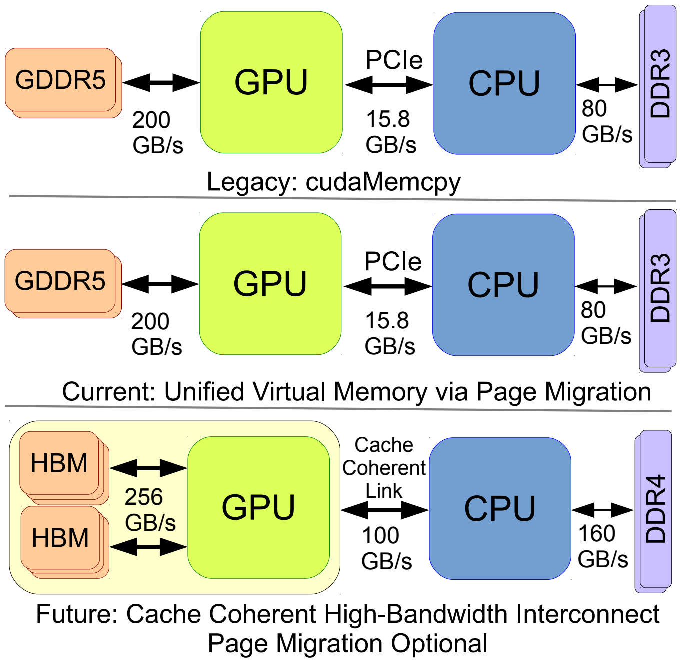


Fig. 1: System architectures for legacy, current, and future mixed GPU-CPU systems.

Recognizing the obstacle this programming model poses to the wider adoption of GPUs in more parallel applications, programming systems like NVIDIA's CUDA, OpenCL, and OpenACC are evolving. Concurrently, GPU-CPU architectures are evolving to have unified globally addressable memory systems in which both the GPU and CPU can access any portion of memory at any time, regardless of its physical location. Today this unified view of memory is layered on top of legacy hardware designs by implementing software-based runtimes that dynamically copy data on demand between the GPU and CPU [1]. As depicted in Figure 1, over the next several years it is expected that GPU and CPU systems will move away from the PCIe interface to a fully cache coherent (CC) interface [2]. These systems will provide high bandwidth and low latency between the non-uniform memory access (NUMA) pools attached to discrete processors by layering coherence protocols on top of physical link technologies such as NVLink [3], Hypertransport [4], or QPI [5]. CC-NUMA access to host memory from the GPU makes the software page migration used today an optional feature thanks to the improved bandwidth, latency, and access granularity that cache coherence can provides.

While interconnect advancements improve GPU-CPU connectivity, no reduction is expected in the memory bandwidth differential between CPU and GPU-attached memory. On-package memories such as High Bandwidth Memory (HBM) or Wide-IO2 (WIO2) may in fact increase this differential as GPU bandwidth requirement continues to grow, feeding the ever increasing number of parallel cores available on GPUs used by both graphics and compute workloads. On the other hand, architects will likely continue to balance latency, power, and cost constraints against raw bandwidth improvement for CPU attached memory, where bandwidth and application performance are less strongly correlated. With application data residing primarily in CPU memory on application start-up, the GPU can access this memory either via hardware cache-coherence (which improves memory system transparency to the programmer) or by migrating a memory page into GPU physical memory (facilitating greater peak bandwidth for future requests). In this work we specifically examine how to best balance accesses through cache-coherence and page migration for a hypothetical CC-NUMA GPU-CPU system connected by a next generation interconnect technology. The contributions of this work are the following:

1) Counter-based metrics to determine when to migrate pages from the CPU to GPU are insufficient for finding an optimal migration policy to exploit GPU memory bandwidth. In streaming workloads, where each page may be accessed only a few times, waiting for N accesses to occur before migrating a page will actually limit the number of accesses that occur after migration, reducing the efficacy of the page migration operation.

2) TLB shootdown and refill overhead can significantly degrade the performance of any page migration policy for GPUs. We show that combining reactive migration with virtual address locality information to aggressively prefetch pages can mitigate much of this overhead, resulting in increased GPU throughput.

3) The legacy intuition to migrate all data to the GPU local memory in an attempt to maximize bandwidth fails to leverage the bandwidth available via the new CC-NUMA interface. A page migration policy which is aware of this differential and balances migration with CC-NUMA link utilization will outperform either GPU or GPU memory being used in isolation.

4) We present a software based memory placement system that, on average, outperforms CC-NUMA based accesses by $1.95\times$, performs 6% better than the legacy CPU to GPU `memcpy` approach by intelligently using both CPU and GPU memory bandwidth, and comes within 28% of oracular page placement, all while maintaining the relaxed memory semantics of modern GPUs.

II. Motivation and Background

A by-product of the GPU's many-threaded design is that it is able to maintain a large number of in-flight memory requests and execution throughput is correlated to memory bandwidth rather than latency, as compared to CPU designs. As a result,

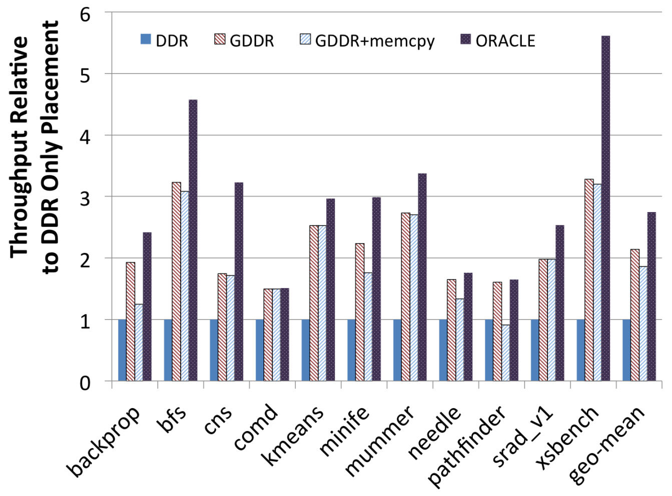


Fig. 2: GPU performance sensitivity to memory subsystem performance where GDDR provides 200GB/s, DDR provides 80GB/s, and `memcpy` bandwidth is 80GB/s.

GPUs have chosen to integrate high bandwidth off-package memory like GDDR rather than accessing the CPU's DDR directly or integrating DDR locally on the GPU board. This choice is motivated by our observation that the performance of some GPU compute workloads would degrade by as much as 66% if the traditional GDDR memory on a GPU were replaced with standard DDR memory, as seen in Figure 2.

In current CPU/GPU designs, GPU and CPU memory systems are private and require explicit copying to the GPU before the application can execute. Figure 2 shows the effect of this copy overhead on application performance by comparing GDDR to GDDR+`memcpy` performance which includes the cost of the programmer manually copying data from the DDR to the GDDR before launching the GPU kernels. While this copy overhead varies from application to application, it can be a non-trivial performance overhead for short-running GPU applications and can even negate the effectiveness of using the high bandwidth GDDR on-board the GPU in a limited number of cases.

While it is technically possible for the GPU to access CPU memory directly over PCIe today, the long latency (microseconds) of the access makes this a rarely used memory operation. Programming system advancements enabling a uniform global address space, like those introduced in CUDA 6.0 [1], relax the requirement forcing programmers to allocate and explicitly copy memory to the GPU up-front, but do nothing to improve the overhead of this data transfer. Further, by copying pages from the CPU to the GPU piece-meal on demand, these new runtimes often introduce additional overhead compared to performing a highly optimized bulk transfer of all the data that the GPU will need during execution. The next step in the evolution of GPUs, given the unified addressing, is to optimize the performance of this new programming model.

A. *Cache Coherent GPUs*

The key advancement expected to enable performance is the introduction of CC-NUMA GPU and CPU systems. Using cache coherence layered upon NVLink, HT, or QPI, GPUs will likely be able to access CPU memory in hundreds of nanoseconds at bandwidths up to 128GB/s by bringing cache lines directly into GPU caches. Figure 2 shows the upper bound (labeled ORACLE) on performance that could be achieved if both the system DDR memory and GPU GDDR memory were used concurrently, assuming data had been optimally placed in both technologies. In this work, we define oracle placement to be *a priori* page placement in the GPU memory (thus requiring no migration), of the minimum number of pages, when sorted from hottest to coldest, such that the GDDR bandwidth is fully subscribed during application execution.

Because initial CPU/GPU CC-NUMA systems are likely to use a form of IOMMU address translation services for walking the OS page tables within the GPU, it is unlikely that GPUs will be able to directly allocate and map their own physical memory without a call back to the CPU and operating system. In this work, we make a baseline assumption that all physically allocated pages are initially allocated in the CPU memory and only the operating system or GPU runtime system executing on the host can initiate page migrations to the GPU. In such a system, two clear performance goals become evident. The first is to design a memory policy that balances CC-NUMA access and page migration to simply achieve the performance of the legacy bulk copy interface without the programming limitations. The second, more ambitious, goal is to exceed this performance and approach the oracular performance by using these memory zones concurrently, enabling a peak memory bandwidth that is the sum of the two zones.

Achieving either of these goals requires migrating enough data to the GPU to exploit its high memory bandwidth while avoiding migrating pages that may never be accessed again. Every page migration increases the total bandwidth requirement of the application and over-migration potentially reduces application performance if sufficient bandwidth headroom in both the DDR and GDDR is not available. Thus, the runtime system must be selective about which pages to migrate. The runtime system also must be cognizant that performing TLB invalidations (an integral part of page migration) on a GPU does not just halt a single processor, but thousands of compute pipelines that may be accessing these pages through a large shared TLB structure. This shared TLB structure makes page migrations between a CPU and GPU potentially much more costly (in terms of the opportunity cost of lost execution throughput) than in CPU-only systems.

In addition to managing the memory bandwidth overhead of page migration and execution stalls due to TLB shootdowns, the relative bandwidth utilization of both the CPU and GPU memory must be taken into account when making page migration decisions. When trying to balance memory bandwidth between two distinct memory zones, it is possible to over- or under-subscribe either memory zone. Migrating pages too slowly to the GPU memory will leave its local memory sitting idle, wasting precious bandwidth. Conversely, migrating pages to the GPU too aggressively may result in under-utilization of the CPU memory while paying the maximum cost in terms of migration overheads. A comprehensive CPU-GPU memory management solution will attempt to balance all of these effects to maximize memory system and GPU throughput in future mobile, graphics, HPC, and datacenter installations.

B. Related Work

Using mixed DRAM technologies or DRAM in conjunction with non-volatile memories to improve power consumption on CPUs has been explored by several groups [6]–[10]. The majority of this work attempts to overcome the performance reductions introduced by non-DDR technologies to improve capacity, power consumption, or both. In CC-NUMA systems, there has been a long tradition of examining where to place memory pages and processes for optimal performance, typically focusing on reducing memory latency [11]–[16]. Whereas CPUs are highly sensitive to memory latency, GPUs can cover a much larger latency through the use of multi-threading. More recent work on page placement and migration [17]–[23] has considered data sharing characteristics, interconnect utilization, and memory controller queuing delays in the context of CPU page placement. However, the primary improvements in many of these works, reducing average memory latency, will not directly apply in a GPU optimized memory system.

Several recent papers have explored hybrid DRAM-NVM GPU attached memory subsystems [24], [25]. Both of these works consider a traditional GPU model where the availability of low latency, high bandwidth access to CPU-attached memory is not considered, nor are the overheads of moving data from the host CPU onto the GPU considered. Several papers propose using a limited capacity, high bandwidth memory as a cache for a larger slower memory [26], [27], but such designs incur a high engineering overhead and would require close collaboration between GPU and CPU vendors that often do not have identically aligned visions of future computing systems.

When designing page migration policies, the impact of TLB shootdown overheads and page table updates is a constant issue. Though most details about GPU TLBs are not public, several recent papers have provided proposals about how to efficiently implement general purpose TLBs that are, or could be, optimized for a GPU's needs [28]–[30]. Others have recently looked at improving TLB reach by exploiting locality within the virtual to physical memory remapping, or avoiding this layer completely [31]–[33]. Finally, Gerofi et al. [34] recently examined TLB performance of the Xeon Phi for applications with large footprints, while McCurdy et al. [35] investigated the effect of superpages and TLB coverage for HPC applications in the context of CPUs.

III. Balancing Page Migration and Cache-Coherent Access

In the future, it is likely GPUs and CPUs will use a shared page table structure while maintaining local TLB caches. It remains to be seen if the GPU will be able to natively walk the operating system page tables to translate virtual to physical address information, or if GPUs will use an IOMMU-like hardware in front of the GPU's native TLB to perform such translations. In either case, the translation from virtual to physical addresses will be implicit, just as it is today for CPUs, and will no longer require trapping back to the CPU to translate or modify addresses. As a result, when page mappings must be modified, all CPUs—and now the GPU—must follow appropriate steps to safely invalidate their local TLB caches. While CPUs typically use a TLB per CPU-core, GPUs use a multi-level global page table across all compute

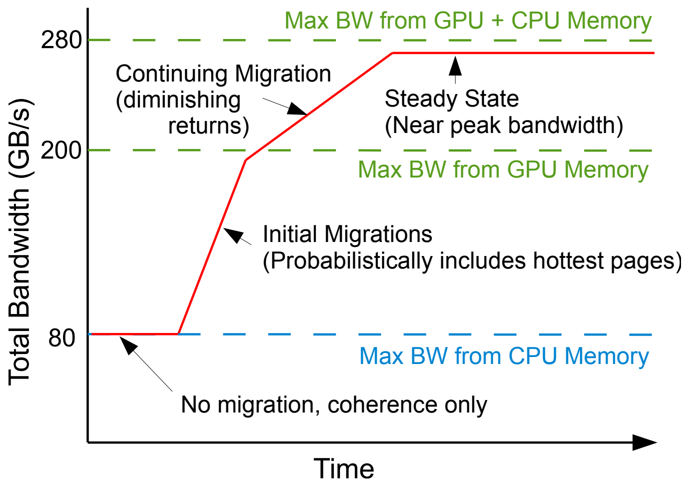


Fig. 3: Opportunity cost of relying on cache coherence versus migrating pages near beginning of application run.

pipelines. Therefore, when TLB shootdowns occur, the penalty will not stall just a single CPU pipeline, it is likely to stall the entire GPU. Whereas recent research has proposed intra-GPU sharer tracking [29] that could mitigate these stalls, this additional hardware is costly and typically unneeded for graphics applications and thus may not be adopted in practice.

Figure 3 provides a visual representation of the effect of balancing memory accesses from both DDR (CPU-attached) and GDDR (GPU-attached) memory. Initially, pages reside entirely in DDR memory. Without migration, the maximum bandwidth available to GPU accesses (via cache coherence to the DDR memory) will be limited by either the interconnect or DDR memory bandwidth. As pages are migrated from DDR to GDDR, the total bandwidth available to the GPU rises as pages can now be accessed concurrently from both memories. Migrations that occur early in kernel execution will have the largest effect on improving total bandwidth, while later migrations (after a substantial fraction of GDDR memory bandwidth is already in use) have less effect. Performance is maximized when accesses are split across both channels in proportion to their peak bandwidth. Figure 3 shows the total bandwidth that is wasted if pages are not migrated eagerly, early in kernel execution. The key objective of the migration mechanism is to migrate the hottest pages as early as possible to quickly ramp up use of GDDR memory bandwidth. Nevertheless, migrating pages that are subsequently never accessed wastes bandwidth on both memory interfaces. In this section, we investigate alternative DDR-to-GDDR memory migration strategies. In particular, we contrast a simple, eager migration strategy against more selective strategies that try to target only hot pages.

A. *Methodology*

To evaluate page migration strategies, we model a GPU with a heterogeneous memory system comprising both GDDR and DDR memories. The GDDR memory is directly attached and addressable by the GPU, as in existing systems. We assume DDR memory may be accessed by the GPU via a cache-line-granularity coherent interface at an additional

Simulator	GPGPU-Sim 3.x
GPU Arch	NVIDIA GTX-480 Fermi-like
GPU Cores	15 SMs @ 1.4Ghz
L1 Caches	16kB/SM
L2 Caches	Memory Side 128kB/DRAM Channel
L2 MSHRs	128 Entries/L2 Slice

Memory system	
GPU-Local GDDR5	8-channels, 200GB/sec aggregate
GPU-Remote DDR4	4-channels, 80GB/sec aggregate
DRAM Timings	RCD=RP=12,RC=40,CL=WR=12
GPU-CPU Interconnect Latency	100 GPU core cycles

TABLE I: Simulation environment and system configuration for mixed memory GPU/CPU system.

100 GPU cycle latency (in addition to the DDR access latency). We derive our latency estimates from SMP CPU interconnects [36]. We find that these additional 100 cycles of latency have relatively little impact on GPU application performance (compared to a hypothetical baseline with no additional latency), as GPUs are already effective in hiding such latency. Across the suite of applications we study, the mean performance degradation due to interconnect latency is 3%, and at worst 10%.

We extend GPGPU-Sim [37] with a model of this GDDR5-DDR4 heterogeneous memory. Table I lists our simulation parameters. We make several additional enhancements to the baseline GTX-480 model to better match the bandwidth requirements of future GPUs (e.g., increasing the number of miss status handling registers, increasing clock frequency, etc). We assume a GDDR bandwidth of 200GB/s and a DDR bandwidth of 80 GB/s.

We model a software page migration mechanism in which migrations are performed by the CPU based on hints provided asynchronously by the GPU. The GPU tracks candidate migration addresses by maintaining a ring buffer of virtual addresses that miss in the GPU TLB. The runtime process on the CPU polls this ring buffer, converts the address to the page aligned base address and initiates migration using the standard Linux `move_pages` system call.

As in a CPU, the GPU TLB must be updated to reflect the virtual address changes that result from migrations. We assume a conventional x86-like TLB shootdown model where the entire GPU is treated like a single CPU using traditional interprocessor interrupt shootdown. In future systems, an IOMMU performing address translations on behalf of the GPU cores is likely to hide the specific implementation details of how it chooses to track which GPU pipelines must be stalled and flushed during any given TLB shootdown. For this work, we make a pessimistic assumption that, upon shootdown, all execution pipelines on the GPU must be flushed before the IOMMU handling the shootdown on behalf of the GPU can acknowledge the operation as complete. We model the time required to invalidate and refill the TLB entry on the GPU as a parameterized, fixed number, of cycles per page migration. In Section III-B we examine the effect of this invalidate/refill overhead on the performance of our migration policy, recognizing that the implementation of TLB structures for GPUs is an active area of research [28], [30].

We model the memory traffic due to page migrations without any special prioritization within the memory controller and rely on the software runtime to rate-limit our migration bandwidth by issuing no more than 4 concurrent page migrations. We study our proposed designs using memory intensive workloads from Rodinia [38] and some other recent HPC applications [39]–[42]. These benchmarks cover varied application domains, including graph-traversal, data-mining, kinematics, image processing, unstructured grid, fluid dynamics and Monte-Carlo transport mechanisms.

B. Results

To understand the appropriate balance of migrating pages early (as soon as first touch on the GPU) or later (when partial information about page hotness is known), we implemented a page migration policy in which pages become candidates for software controlled page migration only after they are touched N times by the GPU. Strictly migrating pages on-demand before servicing the memory requests will put page migration on the critical path for memory load latency. However, migrating a page after N references reduces the number of accesses that can be serviced from the GPU local memory, decreasing the potential impact of page migration. Once a page crosses the threshold for migration, we place it in an unbounded FIFO queue for migration, and allow the CUDA software runtime to migrate the pages by polling this FIFO and migrating pages as described in the previous sub-section.

To isolate the effect of choosing a threshold value from TLB shootdown costs, we optimistically assume a TLB shootdown and refill overhead of 0 cycles for the results shown in Figure 4. This figure shows application performance when migrating pages only after they have been touched N times, represented as threshold-N in the figure. The baseline performance of 1.0 reflects application performance if the GPU only accesses the CPU's DDR via hardware cache coherence and no page migrations to GDDR occur. Although we anticipated using a moderately high threshold (64–128) would generate the best performance (by achieving some level of differentiation between hot and cold data), the results in the figure indicate that, for the majority of the benchmarks, using the lowest threshold typically generates the best performance. Nevertheless, behavior and sensitivity to the threshold varies significantly across applications.

For the majority of our workloads, the best performance comes at a low migration threshold with performance degrading as the threshold increases. The peak performance is well above that achievable with only cache-coherent access to DDR memory, but it rarely exceeds the the performance of the legacy `memcpy` programming practice. The `bfs` benchmark is a notable outlier, with higher migration thresholds improving performance by successfully differentiating hot and cold pages as candidates for migration. However, performance variation due to optimal threshold selection is much smaller than the substantial performance gain of using any migration policy. `Minife` is the second substantial outlier, with a low migration threshold decreasing performance below that of using CPU-only memory, while migration with higher thresholds provides

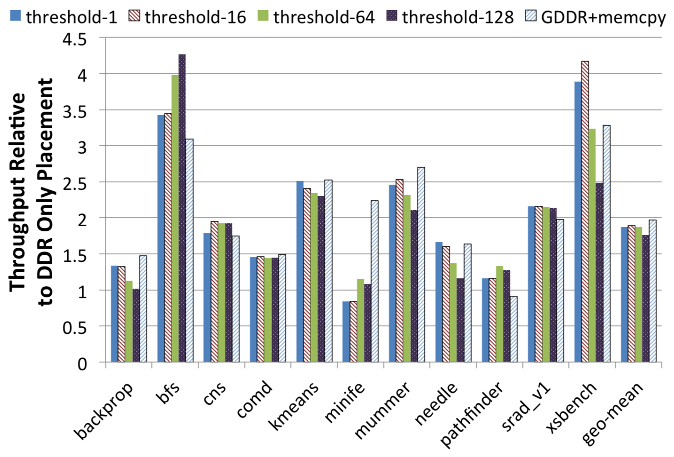


Fig. 4: Performance of applications across varying migration thresholds, where threshold-N is the number of touches a given page must receive before being migrated from CPU-local to GPU-local memory.

only modest gains over cache-coherent access to DDR. Further analysis revealed that, for this workload, migration often occurs after the application has already performed the bulk of its accesses to a given page. In this situation, page migration merely introduces a bandwidth tax on the memory subsystem with little possibility for performance gain.

To implement a threshold-based migration system in practice requires tracking the number of times a given physical page has been touched. Such counting potentially requires tracking all possible physical memory locations that the GPU may access and storing this side-band information either in on-chip SRAMs at the L2, memory controller, or within the DRAM itself. Additional coordination of this information may be required between the structures chosen to track this page-touch information. Conversely, a first touch policy (threshold-1) requires no tracking information and can be trivially implemented by migrating a page the first time the GPU translates an address for the page. Considering the performance differential seen across thresholds, we believe the overhead of implementing the necessary hardware counters to track all pages within a system to differentiate their access counts is not worth the improvement over a vastly simpler first-touch migration policy.

In Figure 4 we showed the performance improvement achievable when modeling the bandwidth cost of the page migration while ignoring the cost of the TLB shootdown, which will stall the entire GPU. At low migration thresholds, the total number of pages migrated is largest and thus application performance is most sensitive to the overhead of the TLB shootdown and refill. Figure 5 shows the sensitivity of application slowdown to the assumed cost of GPU TLB shootdowns for a range of client-side costs similar to those investigated by Villavieja et al. [29]. While the TLB invalidation cost in current GPUs is much higher, due to complex host CPU interactions, it is likely that TLB invalidation cost will drop substantially in the near future (due to IOMMU innovation) to a range competitive with contemporary CPUs (i.e., 100 clock cycles).

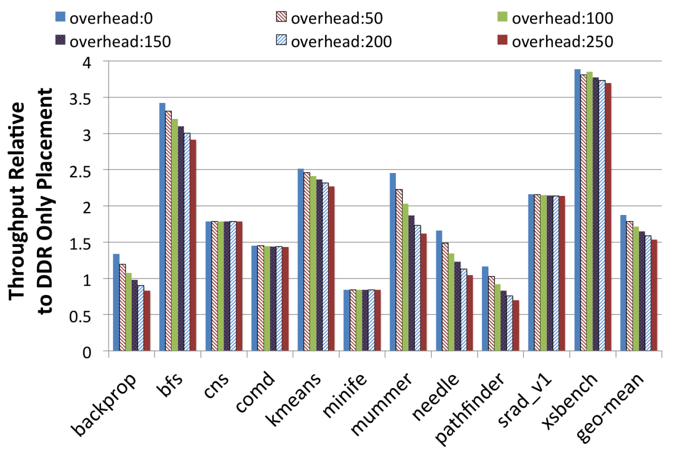


Fig. 5: Performance overhead of GPU execution stall due to TLB shootdowns when using a first touch migration policy (threshold-1).

Because the GPU comprises many concurrently executing pipelines, the performance overhead of a TLB shootdown, which may require flushing all compute pipelines, is high; it may stalls thousands of execution lanes rather than a single CPU core. Figure 5 shows that moving from an idealized threshold of zero, to a realistic cost of one hundred reduces average performance by 16%. In some cases this overhead can negate the entire performance improvement achieved through page migration. To maximize the performance under page migration, our migration mechanism must optimize the trade-off between stalling the GPU on TLB shootdowns versus the improved memory efficiency of migrating pages to the GPU. One way to reduce this cost is to simply perform fewer page migrations, which can be achieved by increasing the migration threshold above the migrate-on-first-touch policy. Unfortunately, a higher migration threshold also decreases the potential benefits of migration. Instead, we will describe mechanisms that can reduce the number of required TLB invalidations simply through intelligent page selection while maintaining the first-touch migration threshold.

IV. Range Expanding Migration Candidates

In the prior section, we demonstrated that aggressively migrating pages generally improves application performance by increasing the fraction of touches to a page serviced by higher-bandwidth (GPU-attached) GDDR versus (CPU-attached) DDR memory. This aggressive migration comes with high overheads in terms of TLB shootdowns and costly GPU pipeline stalls. One reason the legacy application directed `memcpy` approach works well is that it performs both aggressive up-front data transfer to GDDR and does not require TLB shootdowns and stalls. Unfortunately, this requirement for application-directed transfer is not well suited to unified globally addressable memory with dynamic allocation-based programming models. In this section, we discuss a prefetching technique that can help regain the performance benefits of bulk memory copying between private memories, without the associated programming restrictions.

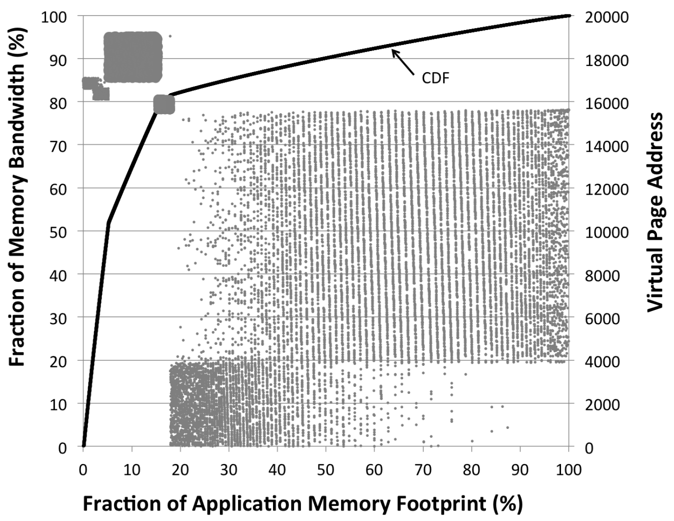
Ideally, a page migration system prefetches pages into GDDR after they are allocated and populated in DDR, but before they are needed on the GPU. Studying the results of the threshold-based migration experiments, we observe that pages often are migrated too late to have enough post-migration accesses to justify the cost of the migration. One way to improve the timeliness of migrations is via a prefetching scheme we call *range expansion*. Range expansion builds on the baseline single-page migration mechanism discussed previously. To implement basic range expansion, when the CUDA runtime is provided a virtual address to be migrated, the runtime also schedules an additional *N* pages in its (virtual address) neighborhood for migration. Pages are inserted into the migration queue in the order of furthest, from the triggered address, to the nearest, to provide the maximum prefetching affect based on spatial locality. We then vary this range expansion amount *N* from 0–128 and discuss the results in Section IV-B.

The motivation for migrating (virtually) contiguous pages can be seen in Figure 6. The figure shows virtual page addresses that are touched by the GPU for three applications in our benchmark set. The X-axis shows the fraction of the application footprint when sampled, after on-chip caches, at 4KB page granularity and sorted from most to fewest accesses. The primary Y-axis (shown figure left) shows the cumulative distribution function of memory bandwidth among the pages allocated by the application. Each point on the secondary scatter plot (shown figure right) shows the virtual address of the corresponding page on the x-axis. This data reveals that hot and cold pages are strongly clustered within the virtual address space. However, the physical addresses of these pages will be non-contiguous due to address interleaving performed by the memory controller. This clustering is key to range expansion because it suggests that if a page is identified for migration, then other neighboring pages in the virtual address space are likely to have a similar number of total touches. To exploit this property, range expansion migrates neighboring virtual addresses of a migration candidate *even if they have not yet been accessed on the GPU*. By migrating these pages before they are touched on the GPU, range expansion effectively prefetches pages to the higher bandwidth memory on the GPU, improving the timeliness and effectiveness of page migrations.

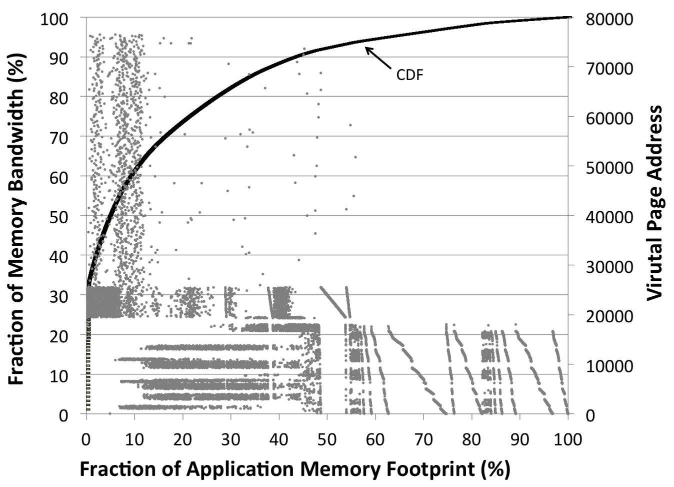
In the case that range expansion includes virtual addresses that are not valid, the software runtime simply allows the `move_pages` system call to fail. This scheme eliminates the need for additional runtime checking or data structure overhead beyond what is already done within the operating system as part of page table tracking. In some cases, a range expansion may extend beyond one logical data structure into another that is laid out contiguously in the virtual address space. While migrating these pages may be sub-optimal from a performance standpoint, there is no correctness issue with migrating these pages to GDDR. For HPC-style workloads with large, coarse-grained memory allocations, this problem happens rarely in practice.

A. Avoiding TLB Shootdowns With Range Expansion

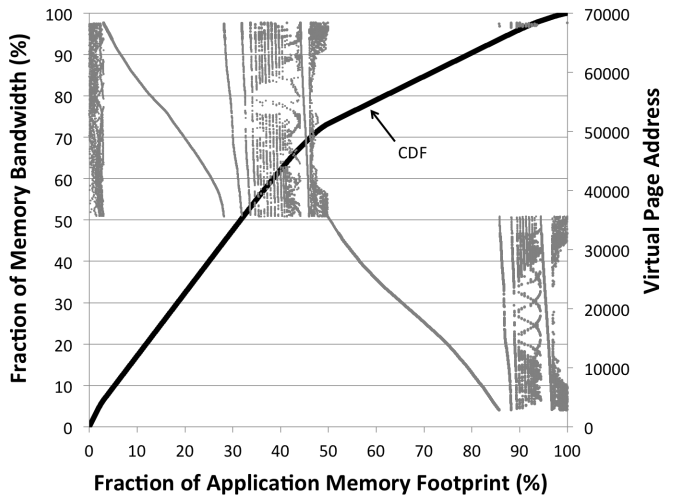
Figure 5 shows that TLB invalidations introduce significant overheads to DDR-to-GDDR migrations. Today, operating systems maintain a list of all processors that have referenced a page so that, upon modification of the page table, TLB shootdowns are only sent to those processor cores (or IOMMU



(a) bfs



(b) xsbench



(c) needle

Fig. 6: Cumulative distribution of memory bandwidth versus application footprint. Secondary axis shows virtual page address of pages touched when sorted by access frequency.

units in the future) that may have a cached translation for this page. While this sharers list may contain false positives, because the mapping entry within a particular sharer may have since been evicted from their TLB, it guarantees that if no request has been made for the page table entry, that core will not receive a TLB shootdown request.

In our previous threshold-based experiments, pages are migrated after the GPU has touched them. This policy has the unfortunate side-effect that all page migrations will result in a TLB shootdown on the GPU. By using range expansion to identify candidates for migration that the GPU is likely to touch but has not yet touched, no TLB shootdown is required (as long as the page is in fact migrated before the first GPU touch). As a result, range expansion provides the dual benefits of prefetching and reducing the number of costly TLB shootdowns.

B. Results

To evaluate the benefits of range expansion, we examine the effect that range expansion has when building on our prior threshold-based migration policies. We considered several thresholds from 1–128 accesses because, while the lowest threshold appears to have performed best in the absence of range expansion, it could be that using a higher threshold, thus identifying only the hottest pages, combined with aggressive range expansion would result in improved performance. We model a fixed TLB shootdown overhead of 100 cycles when performing these experiments, matching the baseline assumptions in the preceding section.

Figure 7 shows application performance as a stacked bar chart on top of the the baseline threshold-based page migration policy for various range expansion values. For the different range expansion values, a single migration trigger is expanded to the surrounding 16, 64, or 128 virtually addressed pages that fall within a single allocation (i.e., were allocated in the same call to `malloc`). The pages identified via range expansion are added to the page migration list in order of furthest, to nearest pages from the triggered virtual address. Pages that are farther from the (already accessed) trigger page are less likely to have been touched by the GPU yet and hence are least likely to be cached in the GPU's TLB. These pages therefore do not require expensive TLB shootdowns and pipeline stalls.

We see that range expansion allows us to outperform not only CC-NUMA access to DDR, but —in many cases— performance exceeds that of the legacy GDDR+`memcpy` implementation. These results indicate that aggressive prefetching, based on first touch access information, provides a balanced method of using both DDR and GDDR memory bandwidth. To understand the improvement from the reduction in TLB shootdowns, we report the fraction of page migrations that required no TLB shootdown in Table II (second column). Compared to threshold-based migrations without range expansion, where all migrations incur a TLB shootdown, range expansion eliminates 33.5% of TLB shootdowns on average and as many as 89% for some applications, drastically reducing the performance impact of these shootdowns.

Figure 7 shows, for `bfs` and `xsbench`, that range expansion provides minimal benefit at thresholds > 1. In these benchmarks, the first touches to contiguous pages are clustered

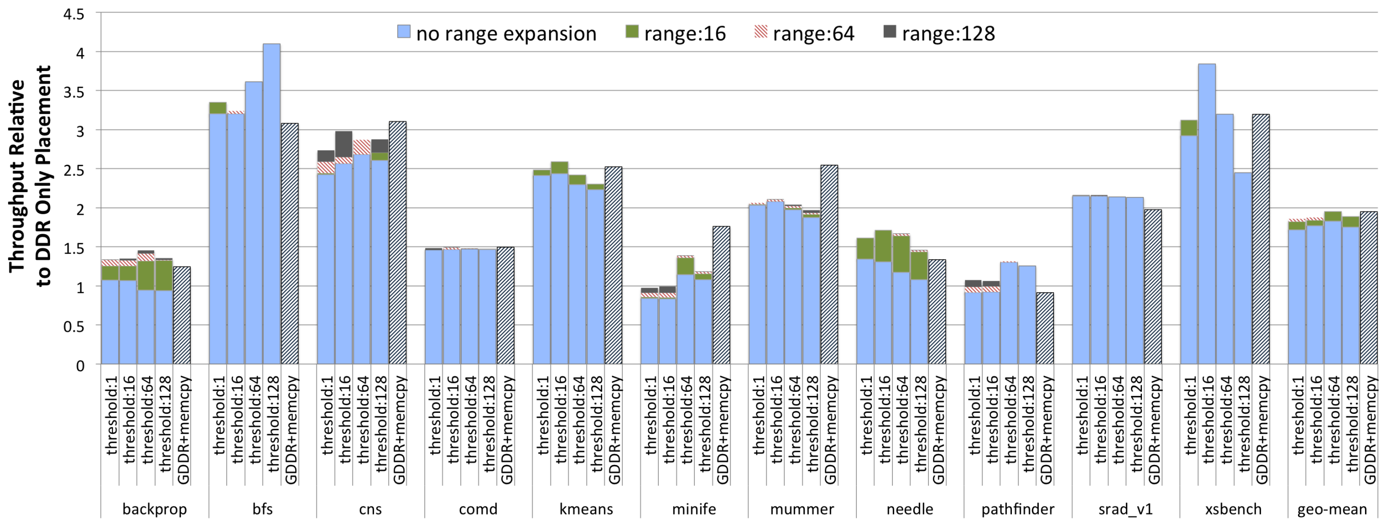


Fig. 7: Effect of range expansion on workload performance when used in conjunction with threshold based migration.

in time, because the algorithms are designed to use blocked access to the key data structures to enhance locality. Thus, the prefetching effect of range expansion is only visible when a page is migrated upon first touch to a neighboring page, by the second access to a page, all its neighbors have already been accessed at least once and there will be no savings from avoiding TLB shootdowns. On the other hand, in benchmarks such as `needle`, there is low temporal correlation among touches to neighboring pages. Even if a migration candidate is touched 64 or 128 times, some of its neighboring pages may not have been touched, and thus the prefetching effect of range expansion provides up to 42% performance improvement even at higher thresholds.

In the case of `backprop`, we can see that higher thresholds perform poorly compared to threshold 1. Thresholds above 64 are simply too high; most pages are not accessed this frequently and thus few pages are migrated, resulting in poor GDDR bandwidth utilization. Range expansion prefetches these low-touch pages to GDDR as well, recouping the performance losses of the higher threshold policies and making them perform similar to a first touch migration policy. For `minife`, previously discussed in subsection III-B, the effect of prefetching via range expansion is to recoup some of the performance loss due to needless migrations. However, performance still falls short of the legacy `memcpy` approach, which in effect, achieves perfect prefetching. Overuse of range expansion hurts performance in some cases. Under the first touch migration policy (threshold-1), using range expansion 16, 64, and 128, the worst-case performance degradations are 2%, 3%, and 2.5% respectively. While not visible in the graph due to the stacked nature of Figure 7, they are included in the geometric mean calculations.

Overall, we observe that even with range expansion, higher-threshold policies do not significantly outperform the much simpler first-touch policy. With threshold 1, the average performance gain with range expansion of 128 is 1.85×. The best absolute performance is observed when using a threshold of 64 combined with a range expansion value of 64, providing 1.95× speedup. We believe that this additional ≈5% speedup over first touch migration with aggressive range expansion is not worth the implementation complexity of tracking and differentiating all pages in the system. In the next section, we discuss how to recoup some of this performance for benchmarks such as `bfs` and `xsbench`, which benefit most from using a higher threshold.

Benchmark	Execution Overhead of TLB Shootdowns	% Migrations Without Shootdown	Excecution Runtime Saved
backprop	29.1%	26%	7.6%
bfs	6.7%	12%	0.8%
cns	2.4%	20%	0.5%
comd	2.02%	89%	1.8%
kmeans	4.01%	79%	3.17%
minife	3.6%	36%	1.3%
mummer	21.15%	13%	2.75%
needle	24.9%	55%	13.7%
pathfinder	25.9%	10%	2.6%
srad_v1	0.5%	27%	0.14%
xsbench	2.1%	1%	0.02%
Average	11.13%	33.5%	3.72%

TABLE II: Effectiveness of range prefetching at avoiding TLB shootdowns and runtime savings under a 100-cycle TLB shootdown overhead.

V. Bandwidth Balancing

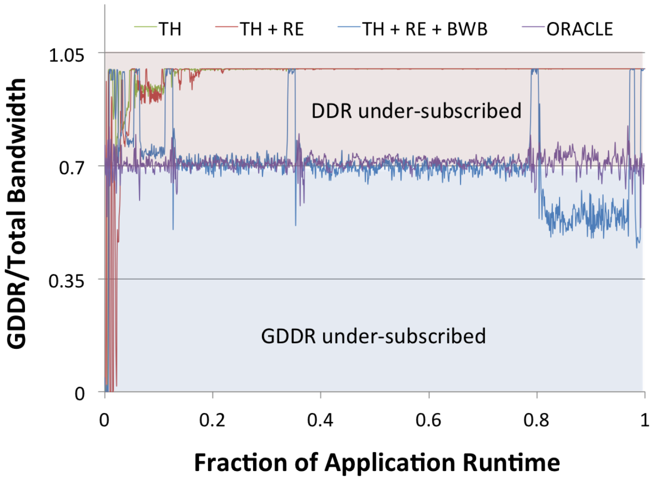
In Section III, we showed that using a static threshold-based page migration policy alone could not ideally balance migrating enough pages to maximize GDDR bandwidth utilization while selectively moving only the hottest data. In Section IV, we showed that informed page prefetching using a low threshold and range expansion to exploit locality within an application's virtual address space matches or exceeds the performance of a simple threshold-based policy. Combining low threshold migration with aggressive prefetching drastically reduces the number of TLB shootdowns at the GPU, reducing the performance overheads of page migration. These policies implemented together, however, will continue migrating pages indefinitely from their initial locations within DDR memory towards the GPU-attached GDDR memory.

As shown in Figure 4, however, rather than migrating all pages into the GPU memory, optimal memory bandwidth utilization is achieved by migrating enough pages to GDDR to maximize its bandwidth while simultaneously exploiting the additional CPU DDR bandwidth via the hardware cache coherence mechanism. To prevent migrating too many pages to GDDR and over-shooting the optimal bandwidth target (70% of traffic to GDDR and 30% to DDR for our system configuration), we implement a migration rate control mechanism for bandwidth balancing. Bandwidth balancing, put simply, allows aggressive migration while the bandwidth ratio of GDDR to total memory bandwidth use is low, and rate limits (or eliminates) migration as this ratio approaches the system's optimal ratio. We implement a simple bandwidth balancing policy based on a sampled moving average of the application's bandwidth needs to each memory type. We assume that the ideal bandwidth ratio in the system can be known either via runtime discovery of the system bandwidth capabilities (using an application like stream [43]) or through ACPI bandwidth information tables, much like memory latency information can be discovered today.

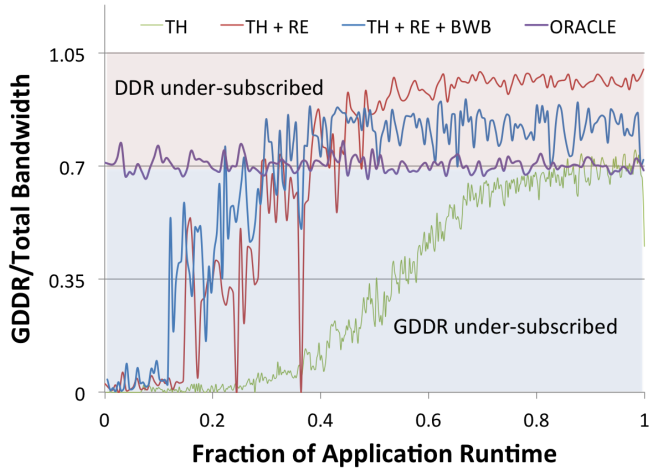
Given the bandwidth capability of each interface, we can calculate the ideal fractional ratio, $GDDR/(DDR{+}GDDR)$, of traffic that should target GDDR using the methodology defined by Agarwal et al. [44]. For the configuration described in Table I, this fraction is 71.4%. We currently ignore command overhead variance between the memory interfaces and assume that it is either the same for technologies in use or that the optimal bandwidth ratio discovered or presented by ACPI will have taken that into account. Using this target, our software page migration samples a bandwidth accumulator present for all memory channels every 10,000 GPU cycles and calculates the average bandwidth utilization of the GDDR and DDR in the system. If this utilization is below the ideal threshold minus 5% we continue migrating pages at full-rate. If the measured ratio approaches within 5% of the target we reduce the rate of page migrations by 1/2. If the measured ratio exceeds the target, we suspend further migrations.

A. Results

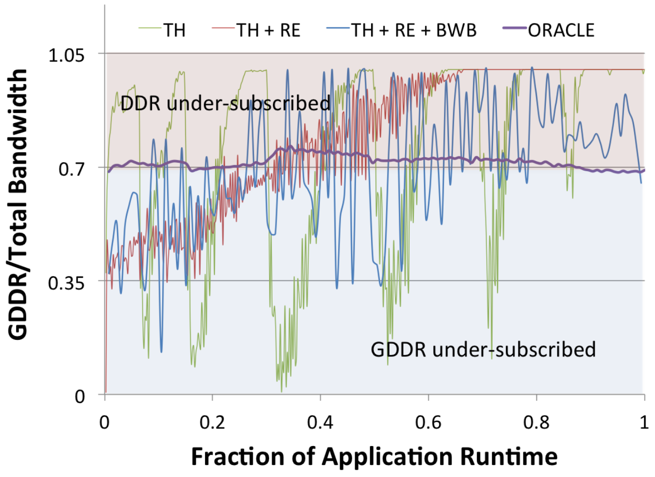
For three example applications, Figure 8 shows the bandwidth utilization of the GDDR versus total bandwidth of the application sampled over time in 1% increments. The TH series provides a view of how migration using single page migration with a static threshold of one (first touch) performs, while $TH + RE$ shows the static threshold with the range expansion solution described in Section IV, and $TH + RE + BWB$ shows this policy with the addition of our bandwidth balancing algorithm. The oracle policy shows that if pages were optimally placed *a priori* before execution there would be some, but not more than 0.1% variance in the GDDR bandwidth utilization of these applications. It is also clear that bandwidth balancing prevents grossly overshooting the targeted bandwidth ratio, as would happen when using thresholds and range expansion alone.



(a) bfs



(b) xsbench



(c) needle

Fig. 8: Fraction of total bandwidth serviced by GDDR during application runtime when when using thresholding alone (TH), then adding range expansion (TH+RE) and bandwidth aware migration (TH+RE+BWB).

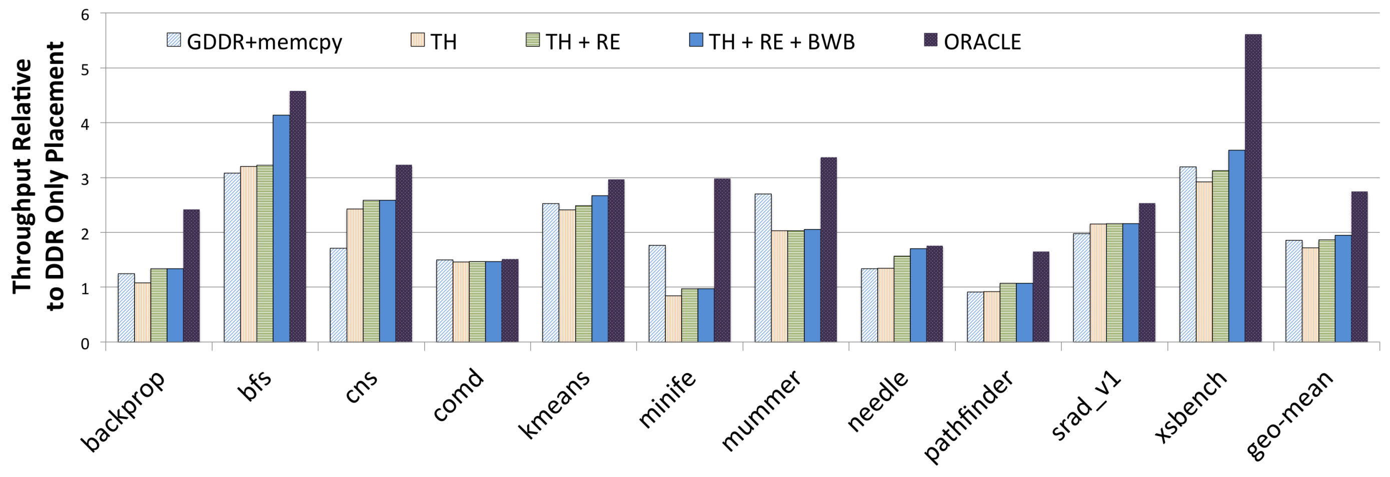


Fig. 9: Application performance when using thresholding alone (TH), thresholding with range expansion (TH+RE), and thresholding combined with range expansion and bandwidth aware migration (TH+RE+BWB).

We investigated various sampling periods shorter and longer than 10,000 cycles, but found that a moderately short window did not cause unwanted migration throttling during the initial migration phase but facilitated a quick adjustment of the migration policy once the target bandwidth balance was reached. If an application's bandwidth utilization subsequently dropped below the target, the short window again enabled rapid reaction to re-enable migration. While there is certainly room for further refinement (e.g., enabling reverse migration when the DDR memory becomes underutilized), our combined solution of threshold-based migration, prefetching via range expansion, and bandwidth balancing is able to capture the majority of the performance available by balancing page migration with CC-NUMA access. Figure 9 shows the results for our implemented solution across our benchmark suite. We see that, on average, we are able to not just improve upon CPU-only DDR by 1.95×, but also exceed the legacy up-front `memcpy`-based memory transfer paradigm by 6%, and achieve 28% of oracular page placement.

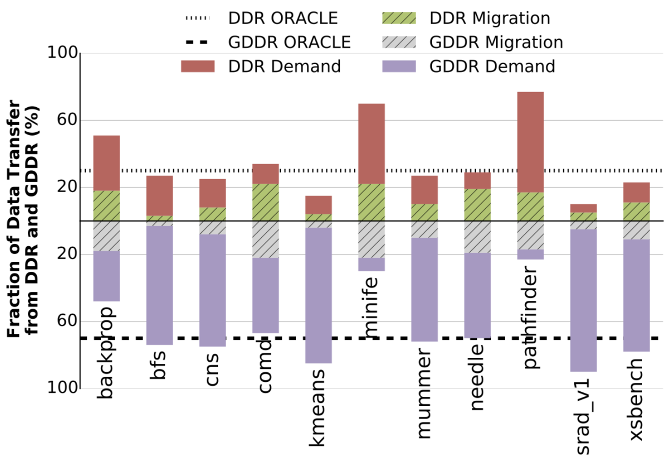


Fig. 10: Distribution of memory bandwidth into demand data bandwidth and migration bandwidth

With our proposed migration policy in place, we seek to understand how it affects the overall bandwidth utilization. Figure 10 shows the fraction of total application bandwidth consumed, divided into four categories. The first, DDR Demand is the actual program bandwidth utilization that occurred via CC-NUMA access to the DDR. The second and third, DDR Migration and GDDR Migration, are the additional bandwidth overheads on both the DDR and GDDR that would not have occurred without page migration. This bandwidth is symmetric because for every read from DDR there is a corresponding write to the GDDR. Finally, GDDR Demand is the application bandwidth serviced from the GDDR. The two additional lines, DDR Oracle and GDDR Oracle, represent the ideal fractional bandwidth that could be serviced from each of our two memories.

We observe that applications which have the lowest GDDR Demand bandwidth see the least absolute performance improvement from page migration. For applications like `minife` and `pathfinder` the GDDR Migration bandwidth also dominates the GDDR Demand bandwidth utilized by the application. This supports our conclusion in subsection III-B that migrations may be occurring too late and our mechanisms are not prefetching the data necessary to make best use of GDDR bandwidth via page migration. For applications that do perform well with page migration, those that perform best tend to have a small amount of GDDR Migration bandwidth when compared to GDDR Demand bandwidth. For these applications, initial aggressive page migration quickly arrives at the optimal bandwidth balance where our bandwidth balancing policy then curtails further page migration, delivering good GDDR Demand bandwidth without large migration bandwidth overhead.

VI. Conclusions

In this work, we have examined a pressing problem that the GPU industry is facing on how to best handle memory placement for upcoming cache coherent GPU-CPU systems. While the problem of page placement in heterogeneous memories has been examined extensively in the context of CPU-only systems, the integration of GPUs and CPUs provides several unique challenges. First, GPUs are extremely sensitive to memory bandwidth, whereas traditional memory placement decisions for CPU-only systems have tried to optimize latency as their first-order concern. Second, while traditional SMP workloads have the option to migrate the executing computation between identical CPUs, mixed GPU-CPU workloads do not generally have that option since the workloads (and programming models) typically dictate the type of core on which to run. This leaves data migration as the only option for co-locating data and processing resources. Finally, to support increasingly general purpose programming models, where the data the GPU shares a common address space with the CPU and is not necessarily known before the GPU kernel launch, programmer-specified up-front data migration is unlikely to be a viable solution in the future.

We have presented a solution to a limited-scope data placement problem for upcoming GPU-CPU systems to enable intelligent migration of data into high bandwidth GPU-attached memory. We identify that demand-based migration alone is unlikely to be a viable solution due to both application variability and the need for aggressive prefetching of pages the GPU is likely to touch, but has not touched yet. The use of range expansion based on virtual address space locality, rather than physical page counters, provides a simple method for exposing application locality while eliminating the need for hardware counters. Developing a system with minimal hardware support is important in the context of upcoming GPU-CPU systems, where multiple vendors may be supplying components in such a system and relying on specific hardware support on either the GPU or CPU to achieve performant page migration may not be feasible. Our migration solution is able to outperform CC-NUMA access alone by 1.95×, legacy application `memcpy` data transfer by 6%, and come within 28% of oracular page placement.

These memory migration policies optimize the performance of GPU workloads with little regard for CPU performance. We have shown that intelligent use of the high bandwidth memory on the GPU can account for as much as a 5-fold performance increase over traditional DDR memory systems. While this is appropriate for applications where GPU performance dominates Amdahl's optimization space, applications with greater data sharing between the CPU and GPU are likely to evolve. Understanding what these sharing patterns look like and balancing the needs of a latency-sensitive CPU versus a bandwidth-hungry GPU is an open problem. Additionally, with memory capacities growing ever larger and huge pages becoming more commonly used, evaluating the trade-off between reducing TLB shootdowns and longer page copy times will be necessary to maintain the high memory bandwidth critical for good GPU performance.

Acknowledgment

The authors would like to thank the numerous anonymous reviewers for their valuable feedback. This work was supported by US Department of Energy and NSF Grant CCF-0845157.

References

[1] NVIDIA Corporation, "Compute Unified Device Architecture," https://developer.nvidia.com/cuda-zone, 2014, [Online; accessed 31-July-2014].

[2] AMD Corporation, "What is Heterogeneous System Architecture (HSA)?" http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/, 2014, "[Online; accessed 28-May-2014]".

[3] NVIDIA Corporation, "NVIDIA Launches World's First High-Speed GPU Interconnect, Helping Pave the Way to Exascale Computing," http://nvidianews.nvidia.com/News/NVIDIA-Launches-World-s-First-High-Speed-GPU-Interconnect-Helping-Pave-the-Way-to-Exascale-Computin-ad6.aspx, 2014, [Online; accessed 28-May-2014].

[4] HyperTransport Consortium, "HyperTransport 3.1 Specification," http://www.hypertransport.org/docs/twgdocs/HTC20051222-0046-0035.pdf, 2010, [Online; accessed 7-July-2014].

[5] INTEL Corporation, "An Introduction to the Intel QuickPath Interconnect," http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009, [Online; accessed 7-July-2014].

[6] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-efficient Main Memory Alternative," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 256–267.

[7] S. Phadke and S. Narayanasamy, "MLP-Aware Heterogeneous Memory System," in *Design, Automation & Test in Europe (DATE)*, March 2011, pp. 1–6.

[8] J. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating System Support for NVM+DRAM Hybrid Main Memory," in *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2009, pp. 14–18.

[9] R. A. Bheda, J. A. Poovey, J. G. Beu, and T. M. Conte, "Energy Efficient Phase Change Memory Based Main Memory for Future High Performance Systems," in *International Green Computing Conference (IGCC)*, July 2011, pp. 1–8.

[10] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *International Conference on Supercomputing (ICS)*, June 2011, pp. 85–99.

[11] K. Wilson and B. Aglietti, "Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C," in *Supercomputing (SC)*, November 2001, pp. 33–35.

[12] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but Effective Techniques for NUMA Memory Management," in *Symposium on Operating Systems Principles (SOSP)*, December 1989, pp. 19–31.

[13] T. Brecht, "On the Importance of Parallel Application Placement in NUMA Multiprocessors," in *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, September 1993, pp. 1–18.

[14] R. LaRowe, Jr., C. Ellis, and M. Holliday, "Evaluation of NUMA Memory Management Through Modeling and Measurements," *IEEE Transactions on Parallel Distribibuted Systems*, vol. 3, no. 6, pp. 686–701, November 1992.

[15] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, September 1996, pp. 279–289.

[16] R. Iyer, H. Wang, and L. Bhuyan, "Design and Analysis of Static Memory Management Policies for CC-NUMA Multiprocessors," *Journal of Systems Architecture*, vol. 48, no. 1, pp. 59–80, September 2002.

[17] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," http://lwn.net/Articles/488709/, 2012, [Online; accessed 29-May-2014].

[18] 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," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013, pp. 381–394.

[19] D. Tam, R. Azimi, and M. Stumm, "Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors," in *European Conference on Computer Systems (EuroSys)*, March 2007, pp. 47–58.

[20] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010, pp. 129–142.

[21] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, no. 3, pp. 54–66, May 2008.

[22] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems," in *USENIX Annual Technical Conference (USENIXATC)*, June 2011, pp. 1–15.

[23] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2010, pp. 319–330.

[24] J. Zhao, G. Sun, G. Loh, and Y. Xie, "Optimizing GPU energy efficiency with 3D die-stacking graphics memory and reconfigurable memory interface," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, pp. 24:1–24:25, December 2013.

[25] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. Vetter, "Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2013, pp. 93–103.

[26] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Integrating DRAM Caches for CMP Server Platforms," *IEEE Micro*, vol. 31, no. 1, pp. 99–108, March 2011.

[27] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, July 2012.

[28] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014, pp. 743–758.

[29] 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 *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2011, pp. 240–249.

[30] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 568–578.

[31] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB Reach using Superpages Backed by Shadow Memory," in *International Symposium on Computer Architecture (ISCA)*, June 1998, pp. 204–213.

[32] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 558–567.

[33] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 237–248.

[34] B. Gerofi, A.Shimada, A. Hori, T. Masamichi, and Y. Ishikawa, "CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores," in *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, June 2014, pp. 73–84.

[35] C. McCurdy, A. Cox, and J. Vetter, "Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008, pp. 95–104.

[36] INTEL Corporation, "Intel Xeon Processor E7-4870," http://ark.intel.com/products/75260/Intel-Xeon-Processor-E7-8893-v2-37_5M-Cache-3_40-GHz, 2014, [Online; accessed 28-May-2014].

[37] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[38] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[39] J. Mohd-Yusof and N. Sakharnykh, "Optimizing CoMD: A Molecular Dynamics Proxy Application Study," in *GPU Technology Conference (GTC)*, March 2014.

[40] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf, "Software Design Space Exploration for Exascale Combustion Co-design," in *International Supercomputing Conference (ISC)*, June 2013, pp. 196–212.

[41] M. Heroux, D. Doerfler, J. Crozier, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories SAND2009-5574, Tech. Rep., 2009.

[42] J. Tramm, A. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, September 2014.

[43] "STREAM - Sustainable Memory Bandwidth in High Performance Computers," http://www.cs.virginia.edu/stream/.

[44] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015.