Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems

Vinson Young[†] [‡], Aamer Jaleel[‡], Evgeny Bolotin[‡], Eiman Ebrahimi[‡], David Nellans[‡], Oreste Villa[‡]

[†]*Georgia Institute of Technology* [‡]*NVIDIA*

vyoung@gatech.edu,{ajaleel,ebolotin,eebrahimi,dnellans,ovilla}@nvidia.com

***Abstract*—Historically, improvement in GPU performance has been tightly coupled with transistor scaling. As Moore's Law slows down, performance of single GPUs may ultimately plateau. To continue GPU performance scaling, multiple GPUs can be connected using system-level interconnects. However, limited inter-GPU interconnect bandwidth (e.g., 64GB/s) can hurt multi-GPU performance when there are frequent remote GPU memory accesses. Traditional GPUs rely on page migration to service the memory accesses from local memory instead. Page migration fails when the page is simultaneously shared between multiple GPUs in the system. As such, recent proposals enhance the software runtime system to replicate read-only shared pages in local memory. Unfortunately, such practice fails when there are frequent remote memory accesses to read-write shared pages. To address this problem, recent proposals cache remote shared data in the GPU last-level-cache (LLC). Unfortunately, remote data caching also fails when the shared-data working-set exceeds the available GPU LLC size.**

This paper conducts a combined performance analysis of state-of-the-art software and hardware mechanisms to improve NUMA performance of multi-GPU systems. Our evaluations on a 4-node multi-GPU system reveal that the combination of work scheduling, page placement, page migration, page replication, and caching remote data still incurs a 47% slowdown relative to an ideal NUMA-GPU system. This is because the shared memory footprint tends to be significantly larger than the GPU LLC size and can not be replicated by software because the shared footprint has read-write property. Thus, we show that existing NUMA-aware software solutions require hardware support to address the NUMA bandwidth bottleneck. We propose *C̲aching R̲emote Data in V̲ideo Me̲mory (CARVE)*, a hardware mechanism that stores recently accessed remote shared data in a dedicated region of the GPU memory. CARVE outperforms state-of-the-art NUMA mechanisms and is within 6% the performance of an ideal NUMA-GPU system. A design space analysis on supporting cache coherence is also investigated. Overall, we show that dedicating only 3% of GPU memory eliminates NUMA bandwidth bottlenecks while incurring negligible performance overheads due to the reduced GPU memory capacity.

***Index Terms*—GPU, Multi-GPU, Memory, NUMA, HBM, DRAM-Cache, Coherence, Page-Migration, Page-Replication**

I. Introduction

GPU acceleration has improved the performance of HPC systems [1], [2], [3], [4] and deep learning applications [5], [6], [7]. Historically, transistor scaling has improved single GPU application performance by increasing the number of Streaming Multiprocessors (SM) between GPU generations. However, the end of Moore's Law [8] necessitates alternative mechanisms, such as multi-GPUs, to continue scaling GPU performance independent of the technology node.

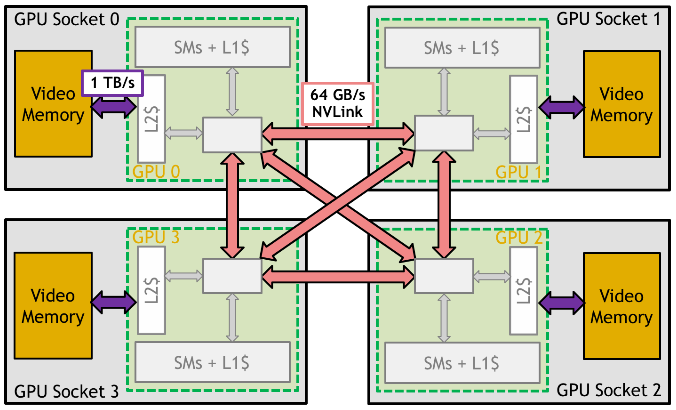

Fig. 1. Multi-GPU System: Low inter-GPU link bandwidth creates Non-Uniform Memory Access (NUMA) bottlenecks.

Multi-GPU systems can speed up application performance by offering 4-8x more computational resources than a single-GPU [9]. Figure 1 illustrates a commercially available multi-GPU system (e.g., DGX [9], [10]) consisting of four GPUs, each with their own dedicated path to high-bandwidth local GPU memory (e.g., 1 TB/s [11]), and interconnected using inter-GPU links (e.g., NVLink [12]). However, due to physical constraints, these inter-GPU links are likely to have *10-20x lower bandwidth* than local GPU memory bandwidth (e.g., 64 GB/s compared to 1 TB/s). Consequently, while multi-GPUs have the potential to provide substantially more compute resources, the large bandwidth discrepancy between local memory bandwidth and inter-GPU bandwidth contributes to Non-Uniform Memory Access (NUMA) behavior that can often bottleneck performance.

The NUMA bandwidth bottleneck is because the local memory is unable to satisfy the majority of memory requests. To address this problem, runtime systems can migrate pages to the local memory. Unfortunately, such practice fails when a page is concurrently accessed by multiple nodes in the system (i.e., shared pages). In such situations, recent proposals enhance the runtime system to *replicate* shared pages [13], [14] in the local memory of each node. However, unbounded replication can significantly increase GPU memory capacity pressure (on average 2.4x in our studies). Since GPUs have limited memory capacity due to memory technology and cost constraints [15], we desire an alternate solution to reduce NUMA bottlenecks.

Recent work investigates software and hardware enhancements to reduce NUMA performance bottlenecks. Specifically,

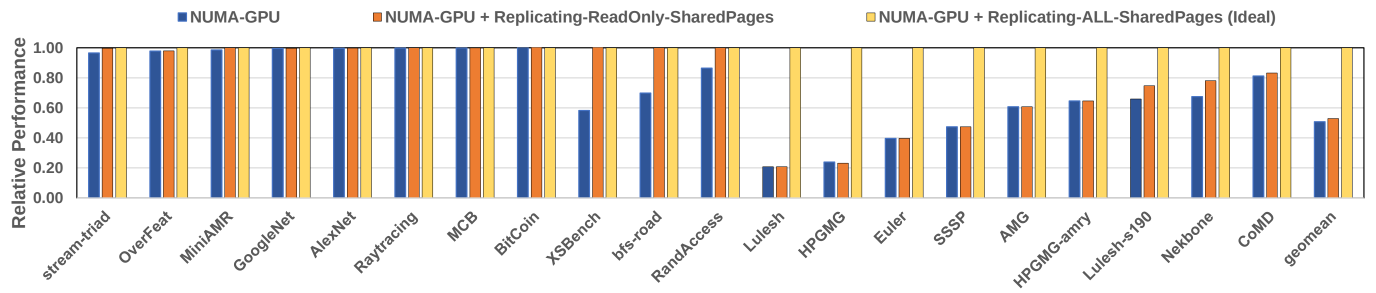

Fig. 2. Performance of NUMA-GPU relative to an ideal paging mechanism that replicates ALL shared pages. There is still a significant performance gap due to limited inter-GPU link bandwidth.

the authors propose work scheduling, first-touch page mapping, and caching shared pages in the GPU cache hierarchy [16]. We refer to this baseline system as *NUMA-GPU*. Figure 2 compares the performance of NUMA-GPU, NUMA-GPU enhanced with replicating read-only shared pages [13], [14] and an *ideal* upper-bound NUMA-GPU system that replicates *all* (both read-only and read-write) shared pages. The x-axis shows the different workloads while the y-axis illustrates performance relative to the ideal NUMA-GPU system that replicates all shared pages. The figure shows that across the 20 workloads studied, eight workloads experience negligible NUMA performance bottlenecks. We also see that replicating read-only shared pages removes NUMA performance bottlenecks for three workloads. For the remainder of the workloads, the baseline NUMA-GPU system experiences 20-80% slowdown that can only be eliminated by replicating read-write shared pages as well.

Read-write pages manifest due to normal program semantics or due to false-sharing when using large pages [17][1]. While read-write shared pages can be replicated, they must be collapsed on writes to ensure data coherence. Unfortunately, the software overhead of collapsing read-write shared pages (even on occasional writes) can be prohibitively high [13]. Consequently, developing efficient software techniques to reduce NUMA performance bottlenecks associated with read-write pages is still an open problem.

Hardware mechanisms that cache shared pages in the GPU cache hierarchy can also reduce the NUMA performance bottleneck [16]. However, conventional GPU LLC sizes are unable to capture the large shared application working-set size of emerging workloads. To address the cache capacity problem, recent papers [19], [20] propose using DRAM caches in multi-node CPU systems. As such, we also investigate the feasibility of architecting DRAM caches in multi-node GPU systems.

This paper investigates the limitations of combining state-of-the-art software and hardware techniques to address the NUMA performance bottleneck in multi-GPU systems. Overall, this paper makes the following contributions:

1) We show that software paging mechanisms commonly used on GPUs today fail to reduce the multi-GPU NUMA performance bottleneck. Furthermore, we also show that caching remote data in the GPU cache hierarchy has limited benefits due to the large shared data footprint. Thus, we show that GPUs must be augmented with large caches to reduce NUMA overheads. To the best of our knowledge, this is the first study that combines state-of-the-art software and hardware techniques (to improve NUMA performance bottlenecks) on a single NUMA platform (i.e., a multi-GPU platform in our study).
2) We propose to augment NUMA-GPU with a practical DRAM cache architecture that increases GPU caching capacity by *Caching Remote Data in Video Memory (CARVE)*. CARVE dedicates a small fraction of the GPU memory to store the contents of remote memory. In doing so, CARVE transforms the GPU memory into a hybrid structure that is simultaneously configured as OS-visible memory and a large cache. CARVE *only* caches remote data in the DRAM cache since there is no latency or bandwidth benefit from caching local data.
3) We perform a detailed design space analysis on the implications of DRAM cache coherence in multi-GPU systems. Specifically, we find that conventional software coherence mechanisms used in GPUs today do not scale to giga-scale DRAM caches. This is because software coherence frequently destroys the data locality benefits from DRAM caches. Instead, we show that GPUs must be extended with a simple hardware coherence mechanism to reap DRAM cache benefits.
4) Finally, we show that the small loss in GPU memory capacity due to CARVE can be compensated by allocating pages in system CPU memory. In such situations, the use of Unified Memory (UM) enables frequently accessed pages to be migrated between system memory and GPU memory. Thus, the performance impact of losing some GPU memory capacity can be tolerable even in situations where CARVE provides limited performance benefits.

Overall, this paper provides an important foundation for further reducing NUMA bottlenecks on emerging multi-GPU platforms. We perform detailed performance evaluations with state-of-the-art work scheduling, software paging, hardware caching, and CARVE on a multi-GPU system consisting of four GPUs. We show that page migration, page replication, and CARVE incur a 49%, 47%, and 6% respective slowdown relative to an ideal NUMA-GPU system that satisfies all memory requests locally. These results show that CARVE is imperative to continue scaling NUMA-GPU performance. CARVE provides these benefits by incurring only a 3% loss in GPU memory capacity. Finally, we show that the loss in GPU memory capacity incurs negligible performance overheads.

[1]GPUs rely on large pages (e.g., 2MB) to ensure high TLB coverage. Reducing the paging granularity to minimize false read-write sharing can cause severe TLB performance bottlenecks [18].

TABLE I
CHARACTERISTICS OF RECENT NVIDIA GPUS.

	Fermi	Kepler	Maxwell	Pascal	Volta
SMs	16	15	24	56	80
BW (GB/s)	177	288	288	720	900
Transistors (B)	3.0	7.1	8.0	15.3	21.1
Tech. node (nm)	40	28	28	16	12
Chip size (mm2)	529	551	601	610	815

II. BACKGROUND AND MOTIVATION

A. GPU Performance Scaling

Table I shows that GPU performance has scaled well over the past decade due to increasing transistor density and GPU memory bandwidth. For example, the recently announced Volta GPU consists of 80 Streaming Multiprocessors (SMs) on a large 815 mm^2 chip consists of 21.1 billion transistors [2]. Unfortunately, the impending end of Moore's Law [8], and limitations in lithography and manufacturing cost [21] threaten continued GPU performance scaling. Without larger or denser dies, GPU architects must now investigate alternative techniques for GPU performance scaling.

Recent advances in packaging [22], signaling [23], [24], and interconnection technology [25], [26] enable new opportunities for scaling GPU performance. For example, recent work propose interconnecting multiple GPUs at a package-level [27] or system-level [16] with high bandwidth interconnection technology (e.g. GRS [26] or NVLink [12]). These multi-GPU systems provide the programmer the illusion of a single GPU system. However, they are non-uniform memory access (NUMA) systems with asymmetric bandwidth to local and remote GPU memory. As such, software and hardware techniques have been proposed to reduce the NUMA bottlenecks. We now discuss these techniques and identify their limitations.

B. NUMA-aware Multi-GPUs

Figure 3 illustrates a NUMA-GPU system where performance is directly dependent on the available remote memory bandwidth. Traditionally, application programmers have manually managed the placement of both compute and data to ensure high NUMA system performance. Such practice significantly increases programmer burden and may not necessarily scale to alternate NUMA systems. To reduce programmer burden, the recent NUMA-GPU [16] proposal presents software and hardware techniques to reduce NUMA bottlenecks, without the need for programmer intervention.

NUMA-GPU enhances the GPU runtime by improving the Cooperative Thread Array (CTA) scheduling and page placement policy. Since adjacent CTAs tend to have significant spatial and temporal locality, NUMA-GPU exploits inter-CTA data locality by scheduling a large batch of contiguous CTAs to each GPU. Furthermore, NUMA-GPU increases the likelihood of servicing the majority of memory requests from the local GPU memory by using a First-Touch (FT) page placement policy. The FT policy allocates pages to the memory of the GPU that first accessed that page. If a page were private, FT mapping ensures that memory accesses are serviced locally.

If a page were already mapped in a remote GPU memory (e.g. shared data), NUMA-GPU caches remote data in the GPU cache hierarchy to enable future references to be serviced at high bandwidth. To ensure correctness, NUMA-GPU extends existing software coherence mechanisms to the GPU LLC [28], [29]. Figure 3 illustrates the software and hardware mechanisms used by NUMA-GPU to reduce NUMA bottlenecks.

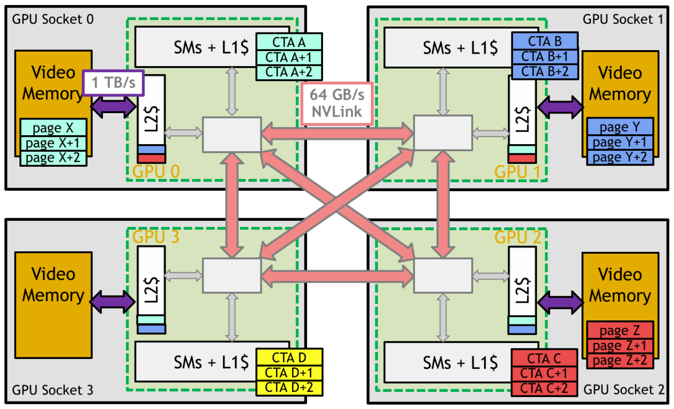

Fig. 3. Organization and bandwidth of a NUMA-Aware Multi-GPU. Distributed CTA-Scheduling and First-Touch Memory Mapping can improve data locality.

C. NUMA-GPU Performance Bottleneck

NUMA systems experience significant performance bottlenecks when applications frequently access remote memory. A recent study showed that remote memory accesses in NUMA systems is because of *false page sharing* when using large page sizes (e.g., to improve TLB coverage [17], [18]). To address the false sharing problem, the runtime system (or OS) can replicate shared pages locally to avoid remote memory accesses. Unfortunately, page replication increases the application memory requirements (on average 2.4x in our study). Thus, page replication works best only when there are free memory pages available. Since future GPU workloads are expected to entirely fill GPU memory capacity [15], page replication is expected to benefit only if the GPU memory is under-utilized.

Besides GPU memory capacity pressure, page replication is only limited to read-only shared pages and not read-write shared pages. This is because the software overhead of collapsing read-write shared pages (even on occasional writes) can be extremely expensive [13]. Thus, page-replication cannot reduce remote memory accesses when a large fraction of remote memory accesses are to read-write shared pages. To illustrate this problem on the baseline NUMA-GPU system, Figure 4 shows the distribution of GPU memory accesses to private pages, read-only shared pages, and read-write shared pages.

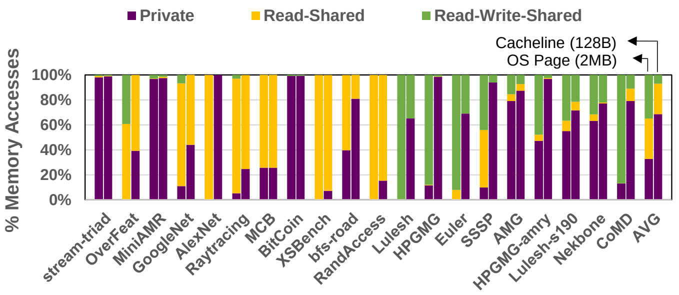

Fig. 4. Distribution of memory accesses to private and shared pages at page size and cache line granularity.

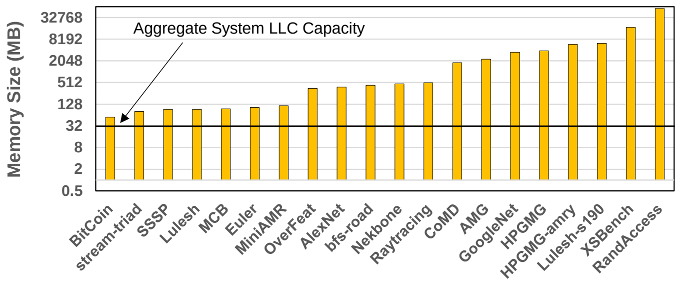

Fig. 5. Total memory capacity (across 4-gpus) needed to cover application shared working set size on a NUMA-GPU.

The figure shows that 40% (up to 100%) of GPU memory accesses are serviced by read-write pages. Consequently, page replication can not reduce NUMA bottlenecks when read-write shared pages are frequently accessed.

To address the limitations of replicating read-write shared pages, NUMA performance can be improved by caching shared data in the on-chip last-level-cache (LLC) [16], [30]. To quantify the total LLC space requirements, Figure 5 shows the shared memory footprint for the different workloads. The shared memory footprint represents the total number of unique remote pages fetched by the different GPUs. The figure clearly shows that caching remote data in the GPU LLC is unlikely to benefit since the application shared memory footprint often significantly exceeds the aggregate system LLC capacity.

A straightforward mechanism to fully cover the shared memory footprint would be to increase the total LLC capacity in the system. However, Figure 5 shows that the system LLC capacity must be drastically increased to fit the shared memory footprint of most workloads. Unfortunately, semiconductor scaling does not allow for such large LLCs to be incorporated on to high-performance processor chips. Consequently, alternative low overhead mechanisms are needed to increase the remote data caching capacity of GPUs.

We investigate increasing the GPU caching capacity by *Caching Remote Data in Video Memory (CARVE)*, a hardware mechanism that statically devotes a small fraction of the local GPU memory to replicate remote shared data. CARVE reduces the granularity of data replication from OS page granularity (i.e., 2MB) to a finer granularity (i.e., 128B cacheline). To ensure correct program execution, however, CARVE requires an efficient mechanism to maintain coherence between multiple GPUs. Thankfully, Figure 4 shows that coherence bandwidth is expected to be small because the fine-grain replication granularity results in a smaller distribution of memory accesses to read-write shared data (further emphasizing the high degree of false-sharing when using large page sizes). Before we investigate increasing GPU caching capacity, we first discuss our experimental methodology.

III. Methodology

We use an industry proprietary trace-driven performance simulator to simulate a multi-GPU system with four GPUs interconnected using high bandwidth links. Each GPU in the system has a processor and memory hierarchy similar to the NVIDIA Pascal GPU [10]. We model 64 SMs per GPU that support 64 warps each. A warp scheduler selects warp instructions each cycle. The GPU memory system consists of a multi-level cache and TLB hierarchy. The first level of the cache and TLB hierarchy are private to each SM, while the last level cache and TLB are shared by all SMs. We assume software-based cache coherence across the private caches that is commonly used in state-of-the-art GPUs [10] today. Table III shows the simulation parameters used in our baseline.

We assume our multi-GPU system is connected using NVLink [12] technology with 64GB/s of bandwidth (in one-direction). We also assume each GPU is interconnected to a CPU using NVLink at 32GB/s. Our infrastructure also includes a detailed DRAM system with 32GB of memory capacity per GPU with 1TB/s of local GPU memory bandwidth. Each GPU memory controller supports 128-entry read and write queues per channel, open-page policy, minimalist address mapping policy [31] and FR-FCFS scheduling policy (prioritizing reads over writes). Writes are issued in batches when the write queue starts to fill up.

We evaluate 20 CUDA benchmarks taken from many applications of interest: HPC applications [32], fluid dynamics [33], graph search [34], machine learning, deep neural networks [4], and medical imaging. We show the memory footprint of our workloads in Table II. We simulate all of our workloads for four billion warp-instructions.

TABLE II
Workload Characteristics

Suite	Benchmark	Abbr.	Mem footprint
HPC	AMG_32	AMG	3.2 GB
	HPGMG-UVM	HPGMG	2.0 GB
	HPGMG-amry-proxy	HPGMG-amry	7.7 GB
	Lulesh-Unstruct-Mesh1	Lulesh	24 MB
	Lulesh-s190	Lulesh-s190	3.7 GB
	CoMD-xyz64_warp	CoMD	910 MB
	MCB-5M-particles	MCB	254 MB
	MiniAMR-15Kv40	MiniAMR	4.4 GB
	Nekbone-18	Nekbone	1.0 GB
	XSBench_17K_grid	XSBench	4.4 GB
	Euler3D	Euler	26 MB
	SSSP	SSSP	42 MB
	bfs-road-usa	bfs-road	590 MB
ML	AlexNet-ConvNet2	AlexNet	96 MB
	GoogLeNet-cudnn-Lev2	GoogLeNet	1.2 GB
	OverFeat-cudnn-Lev3	OverFeat	88 MB
Other	Bitcoin-Crypto	Bitcoin	5.6 GB
	Optix-Raytracing	Raytracing	150 MB
	stream-triad	stream-triad	3.0 GB
	Random Memory Access	RandAccess	15.0 GB

TABLE III
Baseline Multi-GPU System

Number of GPUs	4
Total Number of SMs	256
Max number of warps	64 per SM
GPU frequency	1GHz
OS Page Size	2MB
L1 data cache	128KB per SM, 128B lines, 4 ways
Total L2 cache	32MB, 128B lines, 16 ways
Inter-GPU interconnect	64 GB/s per link, uni-directional
CPU-GPU interconnect	32 GB/s per GPU
Total DRAM bandwidth	4 TB/s
Total DRAM capacity	128GB

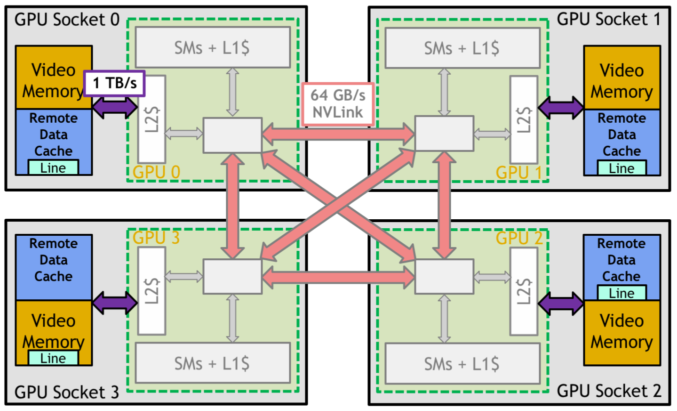

Fig. 6. Architecting a small fraction of GPU memory as a Remote Data Cache (RDC) enables caching large remote data working-sets.

IV. Increasing GPU Cache Capacity

The NUMA bottleneck occurs when a large fraction of memory accesses are serviced by remote memory over low-bandwidth interconnection links [12], [35], [36]. While NUMA has been well studied using conventional memory technology [14], [30], [37], emerging high bandwidth memory technology enables a novel solution space to tackle the NUMA problem. As such, we investigate hardware mechanisms that dedicate a small fraction of local memory to store the contents of remote memory. Doing so enables the remote memory data to be serviced at high local memory bandwidth. In the context of GPUs, this approach effectively increases the caching capacity of a GPU.

A. Caching Remote Data in Video Memory

We propose Caching Remote Data in Video Memory (CARVE), a hardware mechanism that statically reserves a portion of the GPUs on-package local memory to store remote data. Since there is no latency or bandwidth advantage to duplicating local memory data, CARVE only stores remote data in the carve-out. We refer to this memory region as a *Remote Data Cache (RDC)* (see Figure 6). To avoid software overheads, the RDC is entirely hardware-managed and invisible to the programmer and GPU runtime system. The amount of GPU memory dedicated for the RDC can be statically set at system boot time (or kernel launch time if only a single kernel is executing on the GPU). Thus, CARVE transforms the local GPU memory into a hybrid structure that is simultaneously configured both as software visible memory and a hardware-managed cache.

CARVE requires minimal changes to the existing GPU design. On a GPU LLC miss, the GPU memory controller determines whether the missing memory address will be serviced by the local GPU memory or a remote GPU memory. If the request maps to the local GPU memory, the data is fetched directly from local memory and sent to the LLC. Otherwise, the memory controller first checks to see if the data is available in the RDC. In the event of a RDC hit, the data is serviced locally without incurring the bandwidth and latency penalty of a remote GPU memory access. However, in the event of an RDC miss, the missing data is retrieved from the remote GPU memory and returned to the LLC. The missing data is also inserted into the RDC to enable future hits.[2]

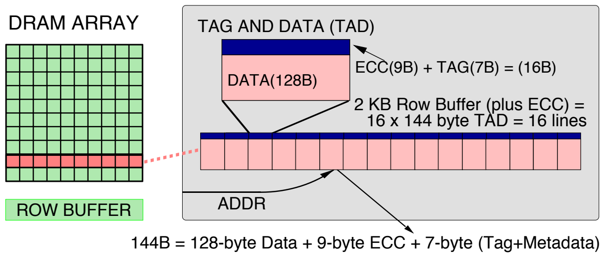
DRAM ARRAY
ROW BUFFER
TAG AND DATA (TAD)
DATA(128B)
ECC(9B) + TAG(7B) = (16B)
2 KB Row Buffer (plus ECC) = 16 x 144 byte TAD = 16 lines
ADDR
144B = 128-byte Data + 9-byte ECC + 7-byte (Tag+Metadata)

Fig. 7. Alloy Cache stores tags with data to enable low latency access. Tags can be stored alongside ECC bits to simplify data alignment and cache controller.

Remote Data Cache (RDC) Design: The RDC is architected at a fine (e.g. 128B) granularity to minimize false sharing (see Figure 4). In our study, we model the RDC as a fine grain Alloy cache [39] (see Figure 7), which organizes its cache as a direct-mapped structure and stores Tags-with-Data. In Alloy, one access to the cache retrieves both the tag and the data. If the tag matches (i.e., a cache hit), the data is then used to service the request. Otherwise, the request is forwarded to the next level of memory. We implement Alloy by storing the tag in spare ECC bits[3], similar to commercial implementation today [40]. Note that RDC is not only limited to the Alloy design and can also be architected using alternate DRAM cache architectures [39], [41], [42].

When GPU memory is configured with an RDC, CARVE requires a single register in the GPU memory controller to specify starting location (i.e., physical address) for the RDC. CARVE also requires simple combinational logic to identify the physical GPU memory location for a given RDC set. Finally, since the RDC is architected in local GPU memory, the different RDC sets are interleaved across all GPU memory channels to ensure high bandwidth concurrent RDC accesses.

Remote Data Cache Evaluation: We evaluate the benefits of CARVE by using a 2GB RDC. This amounts to 6.25% of the baseline 32GB GPU memory dedicated as an RDC and the

[2]The RDC can also store data from System Memory. However, this requires additional support for handling CPU-GPU coherence [38].

[3]We assume GPU memory uses HBM technology. HBM provides 16B of ECC to protect 128B of data. Assuming ECC at a 16-byte granularity (enough to protect bus transfers), SECDED requires 8*9=72 bits to protect the data. This leaves 56 spare ECC bits that can be used for tag and metadata (RDC only requires 6 bits for tags).

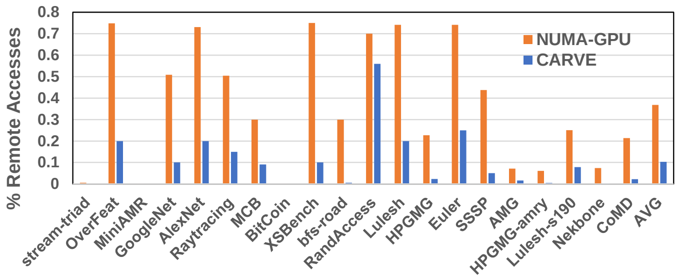

Fig. 8. Fraction of remote memory accesses. CARVE reduces the average fraction of remote memory accesses from 40% in NUMA-GPU to 8%.

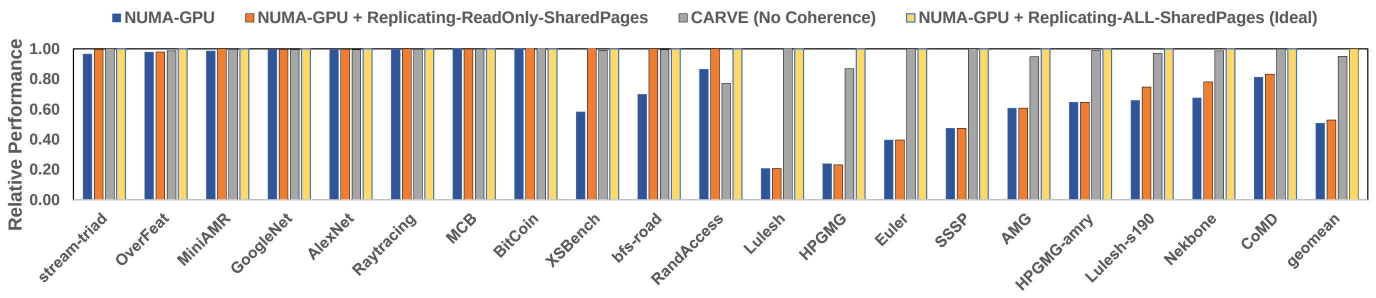

Fig. 9. CARVE performance with zero overhead cache coherence. CARVE is able to achieve Ideal-NUMA-GPU speedup.

remaining 93.75% exposed as operating system (OS) managed memory (30GB per GPU).

RDC Effect on NUMA Traffic: Figure 8 shows the fraction of remote memory accesses with NUMA-GPU and NUMA-GPU enhanced with CARVE. The figure shows that many workloads on the baseline NUMA-GPU system still suffer more than 70% remote memory accesses (e.g., *XSBench* and *Lulesh*). This is because the shared memory footprint of these workloads is larger than the GPU LLC. CARVE on the other hand significantly reduces the fraction of remote memory accesses. On average, NUMA-GPU experiences 40% remote memory accesses while CARVE experiences only 8% remote memory accesses. In doing so, CARVE efficiently utilizes local unused HBM bandwidth. Consequently, the figure shows that CARVE can significantly reduce the inter-GPU bandwidth bottleneck for most workloads.

RDC Performance Analysis (Upper Bound): We first evaluate the performance potential of CARVE by assuming zero-overhead cache coherence between the RDCs (called CARVE-No-Coherence). This enables us to quantify the benefits of servicing remote GPU memory accesses in local GPU memory. We investigate coherence in the next subsection.

Figure 9 shows the performance of NUMA-GPU, NUMA-GPU with software page replication of read-only shared pages, CARVE-No-Coherence, and an ideal NUMA-GPU system that replicates all shared pages. The x-axis shows the different workloads, and the y-axis shows the performance relative to the ideal NUMA-GPU system. The figure shows that CARVE enables workloads like *Lulesh*, *Euler*, *SSSP* and *HPGMG* (which experience significant slowdown with NUMA-GPU and NUMA-GPU enhanced with read-only shared page replication) to approach the performance of an ideal NUMA-GPU system. This is because these workloads experience a 20-60% reduction in remote memory accesses (see Figure 8). This shows that extending the remote data caching capacity into the local GPU memory by using a Remote Data Cache (RDC) significantly reduces the NUMA bandwidth bottleneck. CARVE provides these performance improvements without relying on any software support for page replication.

While CARVE improves performance significantly across many workloads, CARVE can sometimes degrade performance. For example, *RandAccess* experiences a 10% performance degradation due to frequent misses in the RDC. The additional latency penalty of first accessing the RDC then accessing remote memory can degrade performance of latency-sensitive workloads. However, using low-overhead cache hit-predictors [39] can mitigate these performance outliers. Overall, Figure 9 shows that CARVE significantly bridges the performance gap between NUMA-GPU and an ideal NUMA-GPU system. On average, the baseline NUMA-GPU system and NUMA-GPU enhanced with read-only shared page replication experience a 50% performance gap relative to the ideal NUMA-GPU system. On the other hand, CARVE-No-Coherence experiences only a 5% performance gap relative to an ideal NUMA-GPU system that replicates all shared pages. These results show significant performance opportunity from CARVE provided we can efficiently ensure data coherence between the RDCs.

B. Coherence Implications of an RDC in GPUs

CARVE has the potential to significantly improve NUMA-GPU performance by reducing most of the remote memory accesses. However, since each Remote Data Cache (RDC) stores a copy of remote GPU memory, an efficient mechanism is necessary to keep the copies coherent. Fortunately, the GPU programming model provides the programmer with an API to explicitly insert synchronization points to ensure coherent data access. Thus, since software coherence [28] is already supported on conventional GPU systems, we first investigate whether CARVE can be easily extended with software coherence.

Software Coherence

Conventional GPU designs maintain software coherence at kernel boundaries by enforcing two requirements. First, at the end of each kernel call, modified data is written back to memory (i.e., *flush dirty data*). Second, at the beginning of the next kernel invocation, the GPU cores must read the updated values from memory. Conventional GPUs enforce these requirements by implementing a write-through L1 cache that is invalidated at every kernel boundary, and a memory-side last-level cache that is implicitly coherent.

TABLE IV
KERNEL-LAUNCH DELAY UNDER SOFTWARE COHERENCE

	L2 Cache (8MB)	RDC (2GB)
Cache Invalidate	8MB, 16 bank, 1/cycle: **4us**	2GB, 1024GB/s local: **2ms** =>**0ms**
Flush Dirty	8MB, 1024-64GB/s: **8us~128us**	2GB, 64GB/s remote: **32ms** =>**0ms**

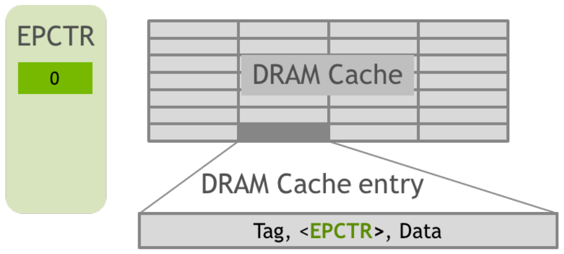

Fig. 10. Epoch-counter based invalidation eliminates the penalty of explicitly invalidating the whole remote data cache.

NUMA-GPUs cache remote data in the GPU LLC and extend software coherence to the GPU LLC by also invalidating the LLC on kernel boundaries [16]. Similarly, RDC coherence can be ensured by further extending software-coherence mechanisms to the RDC. Thus, on every kernel boundary, the memory controller would also invalidate all RDC entries and write back dirty data (if any) to remote GPU memory. Table IV compares the worst case software coherence overhead for RDC and on-chip LLCs. The table shows that invalidating and flushing dirty data in on-chip LLCs tends to be on the order of microseconds and can be tolerated within kernel launch latency. On the other hand, the overhead of invalidating and writing back millions of RDC lines can take on the order of milliseconds. Incurring millisecond latency at every kernel boundary can significantly impact application performance. Consequently, we desire architecture support for efficiently invalidating and writing back RDC lines.

Efficient RDC Invalidation: Since the RDC tag, valid, and dirty bits are all stored in GPU memory, invalidating all RDC lines requires reading and writing GPU memory. Depending on the RDC size, RDC invalidation can be a long latency and bandwidth intensive process. Alternatively, we do not need to physically invalidate all of the lines. We simply need to know if the RDC data is stale (i.e., from a previous kernel and should be re-fetched) or up-to-date (i.e., from current kernel and can be used). If we track the *epoch* an RDC cacheline was installed in, we can easily determine if the RDC data is stale or not. Thus, an epoch-counter-based invalidation scheme is used to invalidate the RDC instantly (i.e., 0 ms) [43], [44].

Figure 10 shows our RDC invalidation mechanism. We use a 20-bit register to maintain an *Epoch Counter (EPCTR)* for each kernel/stream running on the GPU. RDC insertions store the current EPCTR value of the kernel/stream with each RDC cacheline (in the ECC bits along with the RDC Tag). By doing so, RDC hits only occur if the tag matches *and* if the EPCTR stored within the RDC cacheline matches the current EPCTR value of the associated kernel/stream. To support efficient RDC invalidation at kernel boundaries, we simply increment the EPCTR of the appropriate kernel. Thus, the RDC hit logic ensures that the newly launched kernel does not consume a previous kernels' stale data. In the rare event that the EPCTR rolls over on an increment, the memory controller physically resets all RDC cachelines by setting the valid bit and EPCTR to zero. Note that an RDC architected in dense DRAM technology enables large counters per RDC line that otherwise were impractical with similar mechanisms applied to on-chip caches [43], [44].

Efficient RDC Dirty Data Flush: We also desire an efficient mechanism to flush dirty lines to remote GPU memory at kernel boundaries. For a writeback RDC, we would need to know which RDC lines must be flushed. Since our RDC invalidation mechanism no longer requires physically reading all RDC lines, an alternate mechanism is necessary to locate dirty RDC lines. We can use a *dirty-map* [45], which tracks RDC regions that have been written to. On a kernel boundary, the dirty-map can be used to determine which RDC regions should be flushed to remote GPU memory. To avoid the on-chip storage overhead for the dirty-map, we propose a write-through RDC instead. Doing so ensures that dirty data is immediately propagated to remote GPU memory. Note that the write bandwidth to remote GPU memory with a write-through RDC is identical to the baseline NUMA-GPU configuration with no RDC.

Our evaluations with writeback (with dirty-map) and write-through RDC showed that a write-through RDC performs nearly as well (within 1% performance) as a write-back RDC. This is because the vast majority of remote data cached in the line granularity RDC tends to be heavily read-biased (see Figure 4). Consequently, for the remainder of this work, we assume a write-through RDC to enable zero-latency dirty data flush and avoid the complexity of maintaining a write-back RDC.

Software Coherence Results: We now evaluate performance of extending software coherence to the RDC. We refer to this design as *CARVE with Software Coherence (CARVE-SWC)*. Figure 11 compares the performance of CARVE-SWC to CARVE with no coherence overhead (CARVE-No-Coherence). The figure shows that CARVE-SWC enables *XSBench* to perform similar to CARVE-No-Coherence. However, despite eliminating all overheads to maintain software coherence for large caches, CARVE-SWC removes all performance benefits of RDC for the remaining workloads. Since the only difference between CARVE-No-Coherence and CARVE-SWC is the flushing of the RDC between kernels, these results suggest significant inter-kernel data locality that is exploited by CARVE-No-Coherence. While this motivates further research work on improving CARVE-SWC by efficiently prefetching remote data at kernel boundaries, this is out of the scope of this paper. Instead, we now investigate hardware coherence mechanisms that do not require flushing the RDC at kernel boundaries.

Hardware Coherence

CARVE-SWC results reveal that it is important to retain RDC data across kernel boundaries. While software coherence can potentially be relaxed to avoid flushing the RDC, it would require additional complexity for maintaining a software directory to track stale copies of shared data cached remotely and invalidate them explicitly. Since similar functionality is available in conventional hardware coherence mechanisms, we investigate extending the RDC with hardware coherence instead.

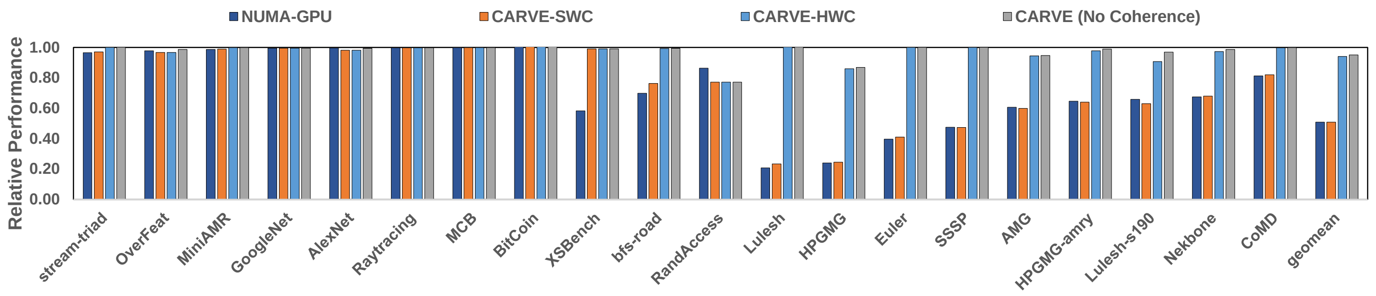

Fig. 11. Performance of CARVE with software coherence and hardware coherence. Software coherence eliminates all of the benefits that CARVE offer. Hardware coherence enables CARVE to reduce NUMA performance bottlenecks.

Since we desire a simple hardware coherence mechanism with minimal design complexity, we augment our RDC design with the simple GPU-VI cache coherence protocol [46].

GPU-VI is a directory-less protocol that implements (a) write-through caches and (b) broadcasts write-invalidates to all remote caches on store requests. Unfortunately, sending a write-invalidate on every store request can significantly increase network traffic in the system. In general, write-invalidates are only required for read-write shared cachelines and can be avoided for private read-write cachelines. Thus, write-invalidate traffic can be reduced by identifying read-write shared cachelines. To that end, we dynamically identify read-write shared cachelines using an *In-Memory Sharing Tracker (IMST)* for *every* memory location at a cacheline granularity. We propose a 2-bit IMST that is stored in the spare ECC space of each cacheline at the *home node*. Figure 12 shows the four possible states tracked by the IMST: uncached, private, read-shared, and read-write shared.

Figure 12 also shows the possible transitions between the different sharing states of a cacheline. These transitions are performed by the GPU memory controller on reads and writes from local and remote GPUs. For example, a remote read transitions the sharing state to *read-shared*. Similarly, a remote write transitions the state to *read-write shared*. Since the IMST tracks the global sharing behavior of a cacheline, a cacheline can perpetually stay in the *read-write shared* or *read-shared* state. To avoid this problem, we probabilistically (e.g., 1%) transition the sharing state to *private* on local GPU writes (after broadcasting invalidates). Note that the IMST differs from conventional cache coherence states like MESI/MOESI which track the instantaneous cacheline state during residency in the cache. The IMST on the other hand monitors the global sharing behavior of a cacheline beyond cache residency.

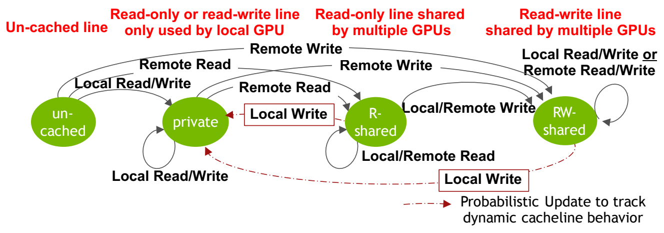
Un-cached line
Read-only or read-write line only used by local GPU
Read-only line shared by multiple GPUs
Read-write line shared by multiple GPUs
Remote Write
Remote Read
Remote Write
Local Read/Write or Remote Read/Write
Local Read/Write
Remote Read
un-cached
private
Local Write
R-shared
Local/Remote Write
RW-shared
Local/Remote Read
Local Read/Write
Local Write
Probabilistic Update to track dynamic cacheline behavior

Fig. 12. In-Memory Sharing Tracker (IMST). Identifying read-write shared cachelines reduces write-invalidate traffic in GPU-VI. The IMST is stored in-memory at a cacheline granularity in spare ECC bits at the home node.

We refer to RDC extended with GPU-VI coherence and IMST as *CARVE with Hardware Coherence (CARVE-HWC)*. CARVE-HWC leverages the IMST to avoid write-invalidates for private cachelines. When the GPU memory controller at the home node receives a write request, it consults the IMST to determine the sharing properties of the cacheline. To avoid bandwidth and latency for reading an IMST entry, we store a copy of the IMST-entry along with the cacheline in the GPU cache hierarchy. Thus, when the IMST-entry identifies the cacheline as private, a write-invalidate broadcast is avoided. However, if the cacheline is detected to be in read-write sharing state, a write-invalidate broadcast is generated. In our work, we find that CARVE-HWC introduces negligible write-invalidate traffic from read-write shared lines since the fine grain RDC enables the majority of memory accesses to reference either private data or read-only shared data (as evident from Figure 4).

Hardware Coherence Results: Figure 11 shows that CARVE-HWC restores RDC benefits lost under CARVE-SWC (e.g., *Lulesh*, *Euler*, and *HPGMG*). In fact, CARVE-HWC nears ideal system performance which replicates all shared pages.

C. CARVE Summary

CARVE significantly improves NUMA-GPU performance by extending the caching capacity of the GPU cache hierarchy. However, CARVE requires an efficient and high-performance cache coherence mechanism to ensure correct data execution. Our investigations in this section show that conventional software coherence mechanisms do not scale with increasing GPU caching capacity. This is because flushing the cache hierarchy between kernel boundaries removes all inter-kernel data locality benefits offered by a high capacity cache hierarchy. Consequently, we find that hardware coherence is necessary to reap the performance benefits of a high capacity GPU cache hierarchy. Our performance evaluations corroborate with results from previous work on GPU L1 caches [47], [48], [49] and conclude that implementing hardware coherence may be a worthwhile endeavor in future NUMA-GPU systems.

V. Results and Analysis

Thus far we have shown CARVE can significantly improve NUMA-GPU performance. However, the performance of NUMA-GPU enhanced with CARVE depends on the Remote Data Cache size, the NUMA-bandwidth differential between

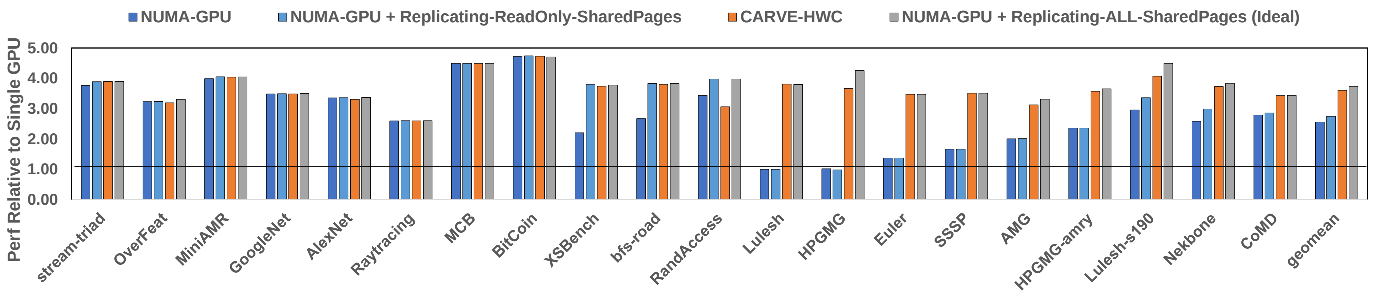

Fig. 13. Performance comparison of CARVE with software support for replicating shared pages. CARVE outperforms read-only shared page replication and nears the performance of an ideal NUMA-GPU system that replicates all pages locally.

local and remote nodes, as well as the potential impact of forcing some application data into the slower system memory (to make room for the Remote Data Cache). This section analyzes the trade-offs associated with these issues, as well as explores the performance sensitivity of our proposals to each of these factors. For the remainder of this section, we only present CARVE-HWC since it performs the best.

A. Results Summary

Figure 13 compares the performance of NUMA-GPU, NUMA-GPU enhanced with page replication of read-only shared pages, NUMA-GPU enhanced with CARVE, and an ideal NUMA-GPU system that replicates all shared pages. The x-axis shows the different workloads while the y-axis represents the performance speedup relative to a single GPU. Overall, we observe that the baseline NUMA-GPU system only achieves a 2.5x speedup relative to a single-GPU system. NUMA-GPU enhanced with software support to replicate read-only shared pages achieves a 2.75x speedup. However, NUMA-GPU enhanced with CARVE outperforms both systems by providing a 3.6x speedup. This is because CARVE services the contents of remote read-only and read-write shared pages from the local GPU memory. In doing so, NUMA-GPU enhanced with CARVE nears the performance of an ideal NUMA-GPU system that provides a 3.7x speedup. These results show that CARVE significantly improves NUMA-GPU performance without relying on any software support for page replication.

B. Sensitivity to Remote Data Cache Size

With any caching proposal, the question of appropriate cache size comes into question, particularly in a proposal like CARVE where the cache capacity could be sized to consume as little or as much of the GPU memory size. Table V(a) sheds light on this question by illustrating the performance sensitivity of CARVE across a variety of remote data cache (RDC) sizes, where NUMA speed-up is multi-GPU speed-up relative to a single-GPU system. We investigate four RDC sizes per GPU: 0.5GB, 1GB, 2GB, and 4GB. On average, we observe that CARVE has minimal sensitivity to RDC size suggesting that dedicating only 1.5% of total GPU memory capacity to remote data caches (i.e. 2GB total out of 128GB total capacity in our system) can improve performance an additional 38% over an existing NUMA-GPU. However, we observed that some workloads like *XSBench*, *MCB*, and *HPGMG* can observe an additional 40-75% speedup when using an aggregate 8GB RDC. This suggests that a runtime mechanism to decide the appropriate RDC size will be an important factor for further multi-GPU performance optimization.

C. Impact of GPU Memory Capacity Loss

NUMA-GPUs enable scaling unmodified GPU applications across multiple GPUs without requiring any programmer involvement. Since the application footprint remains the same despite scaling across multiple GPUs, the aggregate system has an abundance of available on-package GPU memory capacity. Consequently, carving out a small fraction of local GPU memory has no impact on the application's ability to fit inside the memory of the NUMA-GPU. However, when application designers optimize their workloads for NUMA-GPU systems and increase their workload footprints to maximize all available GPU memory capacity, this condition may no longer hold true. When generalizing the CARVE proposal to multi-GPU systems where the application is hand optimized in both application footprint and memory placement, CARVE may in fact force some fraction of the application footprint to spill into system memory, resulting in GPU memory over-subscription that must be handled by software managed runtime systems like NVIDIA's Unified Memory [15].

To quantify this phenomenon, Table V(b) shows the geometric mean slowdown across all workloads when 0%, 1.5%, 3%, 6%, and 12%, of the application memory footprint is placed in system memory under a Unified Memory like policy. We observe that on average, a small GPU memory carve-out (1.5%) has minimal performance degradation (1%) while increased RDC sizes begin to substantially hurt workload throughput. The performance of software based paging systems between system memory and GPU memory is an active area of research with performance improving rapidly [38], [50]. Because software paging between system memory and GPU memory focuses on the cold end of the application data footprint, while CARVE focuses on the hottest shared portion, we believe

TABLE V
PERFORMANCE SENSITIVITY TO RDC SIZE

	Aggregate Memory Carve-Out	(a) NUMA Speed-up (over 1-GPU)	(b) Slowdown due to Carve-Out
NUMA-GPU	0.00%	2.53x	1.00x
CARVE-0.5GB	1.5%	3.50x	0.96x
CARVE-1GB	3.12%	3.55x	0.94x
CARVE-2GB	6.25%	3.61x	0.83x
CARVE-4GB	12.5%	3.65x	0.76x

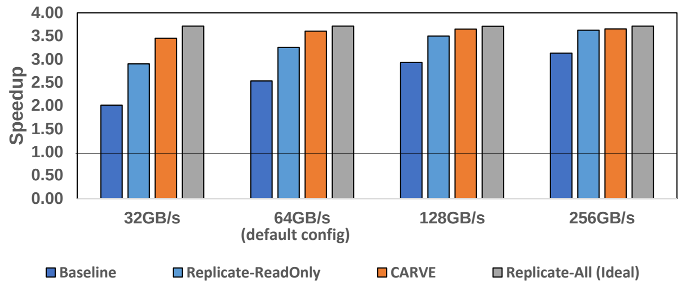

Fig. 14. Performance sensitivity of NUMA-GPU to different inter-GPU link bandwidths normalized to single GPU. We illustrate baseline NUMA-GPU, NUMA-GPU enhanced with replicating read-only shared pages, CARVE, and an ideal system that replicates all shared pages.

these techniques may remain largely orthogonal, though clearly they may have some intersection in implementation.

Taking these trade-offs into account we conclude that CARVE is likely to be a worthwhile trade-off in the face of growing NUMA effects in multi-GPU systems. While CARVE has a high potential for reducing NUMA traffic, CARVE may degrade performance in situations where the entire GPU memory is required by the application. Similar to dynamic caching solutions, better understanding of this trade-off and how rapidly the dynamic balance between these two opposing forces can be balanced will be key to designing an optimal multi-GPU memory system.

D. Sensitivity to Inter-GPU Link Bandwidth

Inter-GPU link bandwidth is one of the largest performance critical factors in future multi-GPU systems. While GPU to GPU connections continue to improve [12], [51], ultimately the link bandwidth between GPUs will always trail the local GPU memory bandwidth. To understand the importance of remote data caching as the ratio of inter-GPU to local memory bandwidth changes, we examine the performance gains of CARVE under a variety of inter-GPU bandwidths in Figure 14. In the figure, x-axis varies the single direction inter-GPU link bandwidth while the y-axis illustrates geometric mean of speedups across all workloads evaluated in this study.

We observe that, the baseline NUMA-GPU performance depends directly on the available inter-GPU link bandwidth. This is because existing NUMA-GPU design (despite caching remote data in the GPU LLC) is unable to entirely capture the shared working set of most workloads. Conversely, CARVE results in a system that is largely insensitive to a range of NUMA-GPU bandwidths, performing very close to an ideal-NUMA-GPU with infinite NUMA bandwidth in all cases. This is because CARVE dedicates local memory space to store remote data, and hence converts nearly all remote accesses to local accesses. In doing so, CARVE eliminates the NUMA bottleneck with a minor decrease in GPU memory capacity.

As multi-GPU systems scale, the NUMA-bandwidth available is unlikely to grow at the same rate as local memory bandwidth. Despite improvements in high speed signaling technology [26], there simply is not enough package perimeter available to enable linear scaling of NUMA bandwidth when doubling or quadrupling the number of closely coupled GPUs in a system. So, while CARVE is still able to improve performance as the ratio of NUMA to local memory bandwidth grows, it is worth noting that CARVE's relative performance will actually increase, should the NUMA link get slower relative to local memory bandwidth (as shown by moving from 64GB/s to 32GB/s links in Figure 14).

E. Scalability of CARVE

NUMA-GPU problems exacerbate as the number of nodes in a multi-GPU system increase. In such situations, CARVE can scale to arbitrary node counts by caching remote shared data locally in the RDC. Since the addition of each GPU node increases total GPU memory capacity, the total memory capacity loss from CARVE is minimal. However, increasing node counts require an efficient hardware coherence mechanism. A directory-less hardware coherence mechanism (like the one used in this paper) can incur significant network traffic overhead for large multi-node systems that experience frequent read-write sharing. In such scenarios, a directory-based hardware coherence mechanism may be more efficient [19], [20]. Depending on the workload sharing behavior, this suggests continued research on efficient hardware coherence mechanisms in multi-node CARVE-enabled GPU systems.

VI. Related Work

Before drawing conclusions we now compare our work to existing work on multi-GPUs, Non-Uniform Memory Access, and giga-scale DRAM cache architectures.

A. Multi-GPUs Systems and Optimization

Multi-GPUs are already used to scale GPU performance for a wide range of workloads [5], [6], [7], [32]. A common method is to use system-level integration [3], [4], [9], [10]. Multi-node GPU systems have also been studied and are employed in the context of high-performance computing and data-center applications [52], [53], [54], [55]. However, programming multi-GPU systems require explicit programmer involvement using software APIs such as Peer-2-Peer access [56] or a combination of MPI and CUDA [57]. This paper explores a transparent Multi-GPU approach that enables a multi-GPU to be utilized as a single GPU (which enables running unmodified single-GPU code) and uses hardware and driver level support to maintain performance [16], [58], [59], [60].

B. Techniques to Reduce NUMA Effects

Existing work has also investigated caching remote data in local caches to reduce NUMA traffic. CC-NUMA [30], S-COMA [14], Reactive NUMA [37] use different mechanisms and granularity for caching remote memory in on-chip caches. CC-NUMA stores remote data in fine-granularity on-chip caches, S-COMA stores remote data at page-granularity in memory with software support, and R-NUMA switches between fine-granularity and page-granularity.

Our work has a similar goal to prior NUMA proposals as we also target reducing remote traffic by storing remote memory contents locally. However, we find that shared working-set sizes are much larger than conventional GPU last-level caches. Unfortunately, semiconductor scaling will not allow for larger on-chip caches to be incorporated into high-performance processor chips. Additionally, we desire to reduce programmer effort in software-based page replication. Our insight is that we can dedicate a portion of the existing GPU memory to store the contents of remote memory. CARVE can be implemented with minimal additional hardware while maintaining programmer and software transparency.

Software page replication can reduce NUMA effects by replicating pages on each remote node [13], [14]. Carrefour [13] proposes to solve NUMA problems with software-based page replication and migration. If accesses are heavily read biased (>99%), it uses page replication to service requests to shared data. If there is heavy memory imbalance, it uses page migration to maintain similar bandwidth out of all memories. While software-based techniques can work, they rely on software support for page protections. Changing page protection of shared pages can incur high latency to collapse the replicated pages [13]. Finally, page replication can significantly reduce GPU memory capacity, which tends to be limited and costly.

C. DRAM Cache Architectures

Emerging high bandwidth memory technology have enabled DRAM caches to be an important research topic [39], [41], [42], [61], [62]. Such work typically targets heterogeneous memory systems where a high bandwidth memory technology (e.g. HBM) is used to cache low bandwidth, denser memory technology (e.g. DDR, PCM, 3DXPoint [63]). Our work on the other hand targets a GPU system that consists of a single memory technology (e.g., HBM, GDDR). Specifically, we transform the conventional GPU memory into a hybrid structure that is simultaneously configured as OS-visible memory as well as a data store for frequently accessed remote shared data.

DRAM Caches have also seen industry application with the High-Bandwidth Cache-Controller (HBCC) in the AMD Vega system [64] and MCDRAM in the Intel Knights Landing (KNL) system [40]. AMD's HBCC also supports video memory to be configured as a page-based cache for system memory. KNL uses a high-bandwidth memory technology called MCDRAM and allows the user to statically configure MCDRAM as cache, memory, or some combination of both. The purpose of HBCC and MCDRAM is to reduce accesses to the next level in the heterogeneous memory hierarchy (e.g., DDR or PCM). While CARVE has design similarities to MCDRAM, CARVE is designed to reduce inter-GPU remote memory accesses and must also consider inter-GPU coherence.

To ensure correct execution of multi-GPU applications, CARVE also requires support for maintaining coherence. Existing approaches CANDY [19] and C3D [20] propose coherent DRAM caches for multi-socket CPU systems. These proposals target maintaining coherence for HBM-based caches in front of traditional DIMM-based memory. CANDY proposes a full coherence directory in the DRAM-cache and caching coherence entries in an on-die coherence directory. C3D proposes a clean DRAM cache to simplify coherence and reduce the latency of coherent memory accesses. CARVE on the other hand extends elements of these existing coherence proposals to a multi-GPU system and also proposes architecture support for extending software coherence.

VII. Conclusions

GPU performance has scaled well over the past decade due to increasing transistor density. However, the slowdown of Moore's Law poses significant challenges for continuous GPU performance scaling. Consequently, multi-GPU systems have been proposed to help meet the insatiable demand for GPU throughput and are commonly used in HPC, data center, and even workstation class machines. For many workloads, multi-GPU execution means taking a leap of faith and re-writing the GPU application to support multi-GPU communications, only then to profile and optimize the application to avoid NUMA performance bottlenecks. With the advent of transparent multi-GPU technology, where a single application is spread across multiple GPUs for execution, NUMA effects on GPUs may impact all GPU workloads, not just those explicitly optimized for NUMA-GPU systems.

NUMA performance bottlenecks are primarily due to frequent remote memory accesses over the low bandwidth interconnection network. Significant research has investigated software and hardware techniques to avoid remote memory accesses. Our evaluations on a multi-GPU system reveal that the combination of page migration, page replication, and caching remote data still incurs significant slowdown relative to an ideal NUMA GPU system. This is because the shared memory footprint tends to be much larger than the GPU LLC and can not be replicated by software because the shared footprint has read-write property. Thus, we show that GPUs must be augmented with large caches to improve NUMA performance. We investigate a hardware mechanism that increases GPU caching capacity by storing recently accessed remote shared data in a dedicated region of the GPU memory. We show that this mechanism can significantly outperform state-of-the-art software and hardware mechanisms while incurring only a minor cost to GPU memory capacity. We then investigate the implications of maintaining GPU cache coherence when increasing the GPU caching capacity. Interestingly, we find that conventional GPU software coherence does not scale to large GPU caches. As such, we show that hardware coherence is necessary to reap the full benefits of increasing GPU caching capacity. Overall, we show that caching remote shared data in a dedicated region of the GPU memory can improve multi-GPU performance within 6% of an ideal multi-GPU system, while incurring negligible performance impact due to the loss in GPU memory capacity.

Acknowledgments

We thank Moinuddin Qureshi for providing helpful organizational comments on an earlier draft of this work. We also thank Sam Duncan, Wishwesh Gandhi, Hemayet Hossain, and Lacky Shah for their feedback over the course of this work.

REFERENCES

[1] InsideHPC, "Top500 shows growing momentum for accelerators," 2015, accessed: 2017-09-19. [Online]. Available: https://insidehpc.com/2015/11/top500-shows-growing-momentum-for-accelerators/

[2] NVIDIA, "Inside Volta: The world's most advanced data center GPU," Jun 2017, accessed: 2017-09-19. [Online]. Available: https://devblogs.nvidia.com/parallelforall/inside-volta/

[3] C.G. Willard, A. Snell, and M. Feldman, "HPC application support for GPU computing," 2015, accessed: 2017-09-19. [Online]. Available: http://www.intersect360.com/industry/reports.php?id=131

[4] O.R.N. Laboratory, "Titan : The world's #1 open science super computer," 2016, accessed: 2017-09-19. [Online]. Available: https://www.olcf.ornl.gov/titan/

[5] NVIDIA, "NVIDIA cuDNN, GPU accelerated deep learning," 2016, accessed: 2017-09-19. [Online]. Available: https://developer.nvidia.com/cudnn

[6] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015, accessed: 2017-09-19. [Online]. Available: http://arxiv.org/abs/1509.09308

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014, accessed: 2017-09-19. [Online]. Available: http://arxiv.org/abs/1409.1556

[8] P. Bright, "Moore's law really is dead this time," 2016, accessed: 2017-09-19. [Online]. Available: http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time

[9] NVIDIA, "NVIDIA DGX-1: The fastest deep learning system," 2017, accessed: 2017-09-19. [Online]. Available: https://devblogs.nvidia.com/parallelforall/dgx-1-fastest-deep-learning-system/

[10] NVIDIA, "NVIDIA Tesla P100 the most advanced datacenter accelerator ever built," 2016, accessed: 2017-09-19. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[11] J. Standard, "High bandwidth memory (HBM) dram," *JESD235A*, 2015. [Online]. Available: https://www.jedec.org/sites/default/files/docs/JESD235A.pdf

[12] NVIDIA, "NVIDIA NVlink high-speed interconnect," 2017, accessed: 2017-09-19. [Online]. Available: http://www.nvidia.com/object/nvlink.html

[13] 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 *ASPLOS '13*. New York, NY, USA: ACM, 2013.

[14] A. Saulasbury, T. Wilkinson, J. Carter, and A. Landin, "An argument for simple coma," in *HPCA '95*, January 1995.

[15] NVIDIA, "Beyond GPU memory limits with unified memory on Pascal," 2016, accessed: 2017-09-19. [Online]. Available: https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal//

[16] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: NUMA-aware GPUs," in *MICRO '17*, Oct 2017.

[17] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema, "Large pages may be harmful on NUMA systems," in *USENIX-ATC '14*. Berkeley, CA, USA: USENIX Association, 2014.

[18] 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 *MICRO '17*, 2017.

[19] C. Chou, A. Jaleel, and M.K. Qureshi, "CANDY: Enabling coherent DRAM caches for multi-node systems," in *MICRO '16*, Oct 2016.

[20] C.C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C3D: Mitigating the NUMA bottleneck via coherent DRAM caches," in *MICRO '16*, Oct 2016.

[21] P. Bright, "The twinscan nxt:1950i dual-stage immersion lithography system," 2016, accessed: 2017-09-19. [Online]. Available: https://www.asml.com/products/systems/twinscan-nxt/twinscan-nxt1950i/en/s46772?dfp_product_id=822

[22] A. Kannan, N.E. Jerger, and G.H. Loh, "Enabling interposer-based disintegration of multi-core processors," in *MICRO '15*, Dec 2015.

[23] M. Mansuri, J.E. Jaussi, J.T. Kennedy, T.C. Hsueh, S. Shekhar, G. Balamurugan, F. O'Mahony, C. Roberts, R. Mooney, and B. Casper, "A scalable 0.128-to-1tb/s 0.8-to-2.6pj/b 64-lane parallel i/o in 32nm cmos," in *ISSCC '13*, Feb 2013.

[24] M. Mansuri, J.E. Jaussi, J.T. Kennedy, T.C. Hsueh, S. Shekhar, G. Balamurugan, F. O'Mahony, C. Roberts, R. Mooney, and B. Casper, "A scalable 0.128-to-1tb/s 0.8-to-2.6pj/b 64-lane parallel i/o in 32nm cmos," in *ISSCC '13*, Feb 2013.

[25] D. Dreps, "The 3rd generation of IBM's elastic interface on POWER6^{TM}," *2007 IEEE Hot Chips 19 Symposium (HCS)*, vol. 00, 2007.

[26] J.W. Poulton, W.J. Dally, X. Chen, J.G. Eyles, T.H. Greer, S.G. Tell, J.M. Wilson, and C.T. Gray, "A 0.54 pj/b 20 gb/s ground-referenced single-ended short-reach serial link in 28 nm cmos for advanced packaging applications," *IEEE Journal of Solid-State Circuits*, vol. 48, Dec 2013.

[27] 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," in *ISCA '17*. New York, NY, USA: ACM, 2017.

[28] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Fermi," 2009, accessed: 2017-09-19. [Online]. Available: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[29] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Kepler gk110," 2012, accessed: 2017-09-19. [Online]. Available: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[30] P. Stenstrom, T. Joe, and A. Gupta, "Comparative performance evaluation of cache-coherent NUMA and COMA architectures," in *ISCA '92*, 1992.

[31] D. Kaseridis, J. Stuecheli, and L.K. John, "Minimalist Open-page: A DRAM page-mode scheduling policy for the many-core era," in *MICRO '11*. New York, NY, USA: ACM, 2011.

[32] L.L.N. Laboratory, "Coral benchmarks," 2014, accessed: 2017-09-19. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[33] 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 *ISSWC '09*. Washington, DC, USA: IEEE Computer Society, 2009.

[34] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS '09*, April 2009.

[35] Intel, "An introduction to the Intel®QuickPath Interconnect," 2009, accessed: 2017-09-19. [Online]. Available: https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

[36] T.H. Consortium, "Meeting the i/o bandwidth challenge: How hypertransport technology accelerates performance in key applications," 2002, accessed: 2017-09-19. [Online]. Available: http://www.opteronics.com/pdf/HTC_WP01_AppsOverview_1001.pdf

[37] B. Falsafi and D.A. Wood, "Reactive NUMA: A design for unifying S-COMA and CC-NUMA," in *ISCA '97*. New York, NY, USA: ACM, 1997.

[38] N. Agarwal, D. Nellans, E. Ebrahimi, T.F. Wenisch, J. Danskin, and S.W. Keckler, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *HPCA '16*, March 2016.

[39] M.K. Qureshi and G.H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *MICRO '12*. IEEE Computer Society, 2012.

[40] A. Sodani, R. Gramunt, J. Corbal, H.S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.C. Liu, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, Mar 2016.

[41] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA '13*. New York, NY, USA: ACM, 2013.

[42] D. Jevdjic, G.H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *MICRO '14*. IEEE, 2014.

[43] X. Yuan, R. Melhem, and R. Gupta, "A timestamp-based selective invalidation scheme for multiprocessor cache coherence," in *ICPP '96*, vol. 3. IEEE, 1996.

[44] S.L. Min and J.L. Baer, "Design and analysis of a scalable cache coherence scheme based on clocks and timestamps," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, 1992.

[45] J. Sim, G.H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *MICRO '12*. IEEE, 2012.

[46] I. Singh, A. Shriraman, W.W.L. Fung, M. O'Connor, and T.M. Aamodt, "Cache coherence for GPU architectures," in *HPCA '13*, 2013.

[47] B.A. Hechtman, S. Che, D.R. Hower, Y. Tian, B.M. Beckmann, M.D. Hill, S.K. Reinhardt, and D.A. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *HPCA '14*, Feb 2014.

[48] M.D. Sinclair, J. Alsop, and S.V. Adve, “Efficient GPU synchronization without scopes: Saying no to complex consistency models,” in *MICRO ’15*, Dec 2015.

[49] J. Alsop, M.S. Orr, B.M. Beckmann, and D.A. Wood, “Lazy release consistency for GPUs,” in *MICRO ’16*, Oct 2016.

[50] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S.W. Keckler, “Towards high performance paged memory for GPUs,” in *HPCA ’16*, March 2016.

[51] NVIDIA, “NVIDIA nvswitch the worlds highest-bandwidth on-node switch,” 2018, accessed: 2018-08-31. [Online]. Available: https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf

[52] Intel, “The Xeon X5365,” 2007, accessed: 2017-09-19. [Online]. Available: http://ark.intel.com/products/30702/Intel-Xeon-Processor-X5365-8M-Cache-3_00-GHz-1333-MHz-FSB

[53] IBM, “IBM zEnterprise 196 technical guide,” 2011, accessed: 2017-09-19. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247833.pdf

[54] IBM, “IBM Power Systems deep dive,” 2012, accessed: 2017-09-19. [Online]. Available: http://www-05.ibm.com/cz/events/febannouncement2012/pdf/power_architecture.pdf

[55] AMD, “AMD server solutions playbook,” 2012, accessed: 2017-09-19. [Online]. Available: http://www.amd.com/Documents/AMD_Opteron_ServerPlaybook.pdf

[56] NVIDIA, “Multi-GPU programming,” 2011, accessed: 2017-09-19. [Online]. Available: http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf

[57] NVIDIA, “MPI solutions for GPUs,” 2016, accessed: 2017-09-19. [Online]. Available: https://developer.nvidia.com/mpi-solutions-gpus

[58] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, “Memory access patterns: the missing piece of the multi-gpu puzzle,” in *SC ’15*, Nov 2015.

[59] J. Cabezas, L. Vilanova, I. Gelado, T.B. Jablin, N. Navarro, and W.m.W. Hwu, “Automatic parallelization of kernels in shared-memory multi-gpu nodes,” in *ICS ’15*. New York, NY, USA: ACM, 2015.

[60] J. Lee, M. Samadi, Y. Park, and S. Mahlke, “Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems,” in *PACT ’13*. Piscataway, NJ, USA: IEEE Press, 2013.

[61] V. Young, P.J. Nair, and M.K. Qureshi, “DICE: Compressing DRAM caches for bandwidth and capacity,” in *ISCA ’17*. New York, NY, USA: ACM, 2017.

[62] V. Young, C. Chou, A. Jaleel, and M.K. Qureshi, “Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction,” in *ISCA ’18*, June 2018.

[63] Intel and Micron, “Intel and Micron produce breakthrough memory technology,” 2015, accessed: 2017-09-19. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

[64] AMD, “Radeon’s next-generation vega architecture,” 2017, accessed: 2017-09-19. [Online]. Available: http://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf