Beyond the Socket: NUMA-Aware GPUs

Ugljesa Milic[†,∓] Oreste Villa[‡] Evgeny Bolotin[‡] Akhil Arunkumar[*] Eiman Ebrahimi[‡]

Aamer Jaleel[‡] Alex Ramirez[⋆] David Nellans[‡]

[†]Barcelona Supercomputing Center (BSC), [∓]Universitat Politècnica de Catalunya (UPC)
[‡]NVIDIA, [*]Arizona State University, [⋆]Google

ugljesa.milic@bsc.es, {ovilla, ebolotin, eebrahimi, ajaleel, dnellans}@nvidia.com, aarunkum@asu.edu,
alexramirez@google.com

ABSTRACT

GPUs achieve high throughput and power efficiency by employing many small single instruction multiple thread (SIMT) cores. To minimize scheduling logic and performance variance they utilize a uniform memory system and leverage strong data parallelism exposed via the programming model. With Moore's law slowing, for GPUs to continue scaling performance (which largely depends on SIMT core count) they are likely to embrace multi-socket designs where transistors are more readily available. However when moving to such designs, maintaining the illusion of a uniform memory system is increasingly difficult. In this work we investigate multi-socket non-uniform memory access (NUMA) GPU designs and show that significant changes are needed to both the GPU interconnect and cache architectures to achieve performance scalability. We show that application phase effects can be exploited allowing GPU sockets to dynamically optimize their individual interconnect and cache policies, minimizing the impact of NUMA effects. Our NUMA-aware GPU outperforms a single GPU by 1.5×, 2.3×, and 3.2× while achieving 89%, 84%, and 76% of theoretical application scalability in 2, 4, and 8 sockets designs respectively. Implementable today, NUMA-aware multi-socket GPUs may be a promising candidate for scaling GPU performance beyond a single socket.

CCS CONCEPTS

• **Computing methodologies** → **Graphics processors**; • **Computer systems organization** → **Single instruction, multiple data**;

KEYWORDS

Graphics Processing Units, Multi-socket GPUs, NUMA Systems

MICRO-50, October 14–18, 2017, Cambridge, MA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124534

ACM Reference format:
Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, David Nellans. 2017. Beyond the Socket: NUMA-Aware GPUs. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3124534

1 INTRODUCTION

In the last decade GPUs computing has transformed the high performance computing, machine learning, and data analytics fields that were previously dominated by CPU-based installations [27, 34, 53, 61]. Many systems now rely on a combination of GPUs and CPUs to leverage high throughput data parallel GPUs with latency critical execution occurring on the CPUs. In part, GPU-accelerated computing has been successful in these domains because of native support for data parallel programming languages [24, 40] that reduce programmer burden when trying to scale programs across ever growing data sets.

Nevertheless, with GPUs nearing the reticle limits for maximum die size and the transistor density growth rate slowing down [5], developers looking to scale the performance of their single GPU programs are in a precarious position. Multi-GPU programming models support explicit programming of two or more GPUs, but it is challenging to leverage mechanisms such as Peer-2-Peer access [36] or a combination of MPI and CUDA [42] to manage multiple GPUs. These programming extensions enable programmers to employ more than one GPU for high throughput computation, but require re-writing of traditional single GPU applications, slowing their adoption.

High port-count PCIe switches are now readily available and the PCI-SIG roadmap is projecting PCIe 5.0 bandwidth to reach 128 GBs in 2019 [6]. At the same time, GPUs are starting to expand beyond the traditional PCIe peripheral interface to enable more efficient interconnection protocols between both GPUs and CPUs, such as AMD's Infinity Fabric or NVIDIA's Scalable Link Interface and NVLink [2, 29, 35, 39, 44]. Future high bandwidth GPU to GPU interconnects, possibly using improved communication protocols, may lead to system designs with closely coupled groups of GPUs that can efficiently share memory at fine granularity.

The onset of such multi-socket GPUs would provide a pivot point for GPU and system vendors. On one hand, vendors can continue to

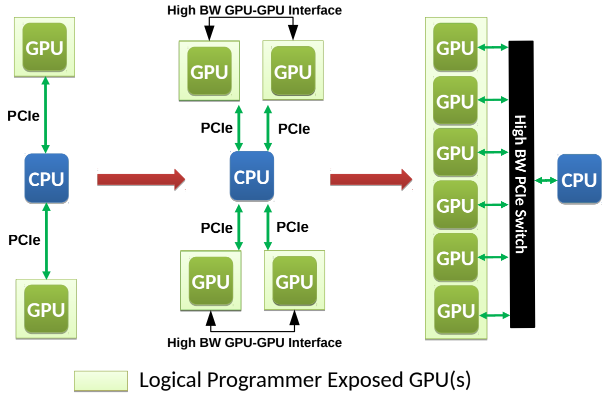


Figure 1: The evolution of GPUs from traditional discrete PCIe devices to single logical, multi-socketed accelerators utilizing a switched interconnect.

expose these GPUs as individual GPUs and force developers to use multiple programming paradigms to leverage these multiple GPUs. On the other, vendors could expose multi-socket designs as a single non-uniform memory access (NUMA) GPU resource as shown in Figure 1. By extending the single GPU programming model to multi-socket GPUs, applications can scale beyond the bounds of Moore's law, while simultaneously retaining the programming interface to which GPU developers have become accustomed.

Several groups have previously examined aggregating multiple GPUs together under a single programming model [7, 30]; however this work was done in an era where GPUs had limited memory addressability and relied on high latency, low bandwidth CPU-based PCIe interconnects. As a result, prior work focused primarily on improving the multi-GPU programming experience rather than achieving scalable performance. Building upon this work, we propose a multi-socket *NUMA-aware* GPU architecture and runtime that aggregates multiple GPUs into a single programmer transparent logical GPU. We show that in the the era of unified virtual addressing [37], cache line addressable high bandwidth interconnects [39], and dedicated GPU and CPU socket PCB designs [29], scalable multi-GPU performance may be achievable using existing single GPU programming models. This work makes the following contributions:

- We show that traditional NUMA memory placement and scheduling policies are not sufficient for multi-socket GPUs to achieve performance scalability. We then demonstrate that inter-socket bandwidth will be the primary performance limiter in future NUMA GPUs.
- By exploiting program phase behavior we show that inter-socket links (and thus bandwidth) should be dynamically and adaptively reconfigured at runtime to maximize link utilization. Moreover, we show that link policy must be determined on a per-GPU basis, as global policies fail to capture per-GPU phase behavior.
- We show that both the GPU L1 and L2 caches should be made NUMA-aware and dynamically adapt their caching policy to minimize NUMA effects. We demonstrate that in NUMA GPUs, extending existing GPU cache coherence protocols across multiple sockets is a good design choice, despite the overheads.
- We show that multi-socket NUMA-aware GPUs can allow traditional GPU programs to scale efficiently to as many as 8 GPU sockets, providing significant headroom before developers must re-architect applications to obtain additional performance.

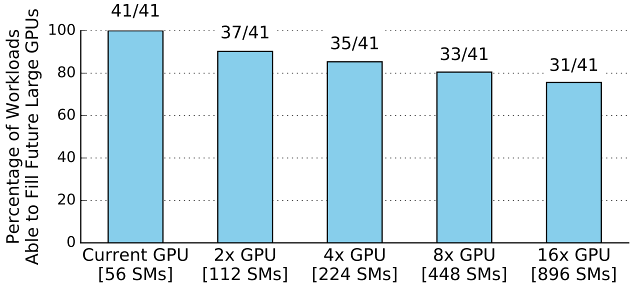


Figure 2: Percentage of workloads that are able to fill future larger GPUs (average number of concurrent thread blocks exceeds number of SMs in the system).

2 MOTIVATION AND BACKGROUND

Over the last decade single GPU performance has scaled thanks to a significant growth in per-GPU transistor count and DRAM bandwidth. For example, in 2010 NVIDIA's Fermi GPU integrated 1.95B transistors on a 529 mm^2 die, with 180 GBs of DRAM bandwidth [13]. In 2016 NVIDIA's Pascal GPU contained 12B transistors within a 610 mm^2 die, while relying on 720 GBs of memory bandwidth [41]. Unfortunately, transistor density is slowing significantly and integrated circuit manufacturers are not providing roadmaps beyond 7 nm. Moreover, GPU die sizes, which have been also slowly but steadily growing generationally, are expected to slow down due to limitations in lithography and manufacturing cost.

Without either larger or denser dies, GPU architects must turn to alternative solutions to significantly increase GPU performance. Recently 3D die-stacking has seen significant interest due to its successes with high bandwidth DRAM [23]. Unfortunately 3D die-stacking still has a significant engineering challenges related to power delivery, energy density, and cooling [60] when employed in power hungry, maximal die-sized chips such as GPUs. Thus we propose GPU manufacturers are likely to re-examine a tried and trued solution from CPU world, *multi-socket GPUs*, to scaling GPU performance while maintaining the current ratio of floating point operations per second (FLOPS) and DRAM bandwidth.

Multi-socket GPUs are enabled by the evolution of GPUs from external peripherals to central computing components, considered at system design time. GPU optimized systems now employ custom PCB designs that accommodate high pin count socketed GPU modules [35] with inter-GPU interconnects resembling QPI or HyperTransport [17, 21, 39]. Despite the rapid improvement in hardware capabilities, these systems have continued to expose the GPUs provided as individually addressable units. These multi-GPU systems can provide high aggregate throughput when running multiple concurrent GPU kernels, but to accelerate a single GPU workload they require layering additional software runtimes on top of native GPU programming interfaces such as CUDA or OpenCL [24, 38]. Unfortunately, by requiring application re-design many workloads are never ported to take advantage of multiple GPUs.

Extending single GPU workload performance significantly is a laudable goal, but we must first understand if these applications will be able to leverage larger GPUs. Assuming that the biggest GPU in the market today amasses ≈50 SMs (i.e. NVIDIA's Pascal GPU contains 56), Figure 2 shows that across a benchmark set of 41 applications that are later described in Section 3.2, most single GPU optimized workloads already contain sufficient data parallelism to fill GPUs that are 2–8× larger than today's biggest GPUs. For those that do not, we find that the absolute number of thread blocks (CTAs) is intentionally limited by the programmer, or that the problem dataset can not be easily scaled up due to memory limitations. While most of these applications that do scale are unlikely to scale to thousands of GPUs across an entire data center without additional developer effort, moderate programmer transparent performance scalability will be attractive for applications that already contain sufficient algorithmic and data parallelism.

In this work, we examine the performance of a future 4-module NUMA GPU to understand the effects that NUMA will have when executing applications designed for UMA GPUs. We will show (not surprisingly), that when executing UMA-optimized GPU programs on a multi-socket NUMA GPU, performance does not scale. We draw on prior work and show that before optimizing GPU microarchitecture for NUMA-awareness, several software optimizations must be in place to preserve data locality. Alone these SW improvements are not sufficient to achieve scalable performance however and interconnect bandwidth is a significant hindrance on performance. To overcome this bottleneck we propose two classes of improvements to reduce the observed NUMA penalty.

First, in Section 4 we examine the ability of switch connected GPUs to dynamically repartition their ingress and egress links to provide asymmetric bandwidth provisioning when required. By using existing interconnects more efficiently, the effective NUMA bandwidth ratio of remote memory to local memory decreases, improving performance. Second, to minimize traffic on oversubscribed interconnect links we propose GPU caches need to become NUMA-aware in Section 5. Traditional on-chip caches are optimized to maximize overall hit rate, thus minimizing off-chip bandwidth. However, in NUMA systems, not all cache misses have the same relative cost and thus should not be treated equally. Due to the NUMA penalty of accessing remote memory, we show that performance can be maximized by preferencing cache capacity (and thus improving hit rate) towards data that resides in slower remote NUMA zones, at the expense of data that resides in faster local NUMA zones. To this end, we propose a new NUMA-aware cache architecture that dynamically balances cache capacity based on memory system utilization. Before diving into microarchitectural details and results, we first describe the locality-optimized GPU software runtime that enables our proposed NUMA-aware architecture.

3 A NUMA-AWARE GPU RUNTIME

Current GPU software and hardware is co-designed together to optimize throughput of processors based on the assumption of uniform memory properties within the GPU. Fine grained interleaving of memory addresses across memory channels on the GPU provides implicit load balancing across memory but destroys memory locality. As a result, thread block scheduling policies need not be sophisticated to capture locality, which has been destroyed by the memory system layout. For future NUMA GPUs to work well, both system software and hardware must be changed to achieve both functionality and performance. Before focusing on architectural changes to build a NUMA-aware GPU we describe the GPU runtime system we employ to enable multi-socket GPU execution.

Prior work has demonstrated feasibility of a runtime system that transparently decomposes GPU kernels into sub-kernels and executes them on multiple PCIe attached GPUs in parallel [7, 25, 30]. For example, on NVIDIA GPUs this can be implemented by intercepting and remapping each kernel call, GPU memory allocation, memory copy, and GPU-wide synchronization issued by the CUDA driver. Per-GPU memory fences must be promoted to system level and seen by all GPUs, and sub-kernels CTA identifiers must be properly managed to reflect those of the original kernel. Cabezas et al. solve these two problems by introducing code annotations and an additional source-to-source compiler which was also responsible for statically partitioning data placement and computation [7].

In our work, we follow a similar strategy but without using a source-to-source translation. Unlike prior work, we are able to rely on NVIDIA's Unified Virtual Addressing [37] to allow dynamic placement of pages into memory at runtime. Similarly, technologies with cache line granularity interconnects like NVIDIA's NVLink [39] allow transparent access to remote memory without the need to modify application source code to access local or remote memory addresses. Due to these advancements, we assume that through dynamic compilation of PTX to SASS at execution time, the GPU runtime will be able to statically identify and promote system wide memory fences as well as manage sub-kernel CTA identifiers.

Current GPUs perform fine-grained memory interleaving at a sub-page granularity across memory channels. In a NUMA GPU this policy would destroy locality and result in 75% of all accesses to be to remote memory in a 4 GPU system, an undesirable effect in NUMA systems. Similarly, a round-robin page level interleaving could be utilized, similar to the Linux interleave page allocation strategy, but despite the inherent memory load balancing, this still results in 75% of memory accesses occurring over low bandwidth NUMA links. Instead we leverage UVM page migration functionality to migrate pages on-demand from system memory to local GPU memory as soon as the first access (also called first-touch allocation) is performed as described by Arunkumar et. al [3].

On a single GPU, fine-grained dynamic assignment of CTAs to SMs is performed to achieve good load balancing. Extending this policy to a multi-socket GPU system is not possible due to the relatively high latency of passing sub-kernel launches from software to hardware. To overcome this penalty, the GPU runtime must launch a block of CTAs to each GPU-socket at a course granularity. To encourage load balancing, each sub-kernel could be comprised of an interleaving of CTAs using modulo arithmetic. Alternatively a single kernel can be decomposed into N sub-kernels, where N is the total number of GPU sockets in the system, assigning equal amount of contiguous CTAs to each GPU. This design choice potentially exposes workload unbalance across sub-kernels, but it also preserves

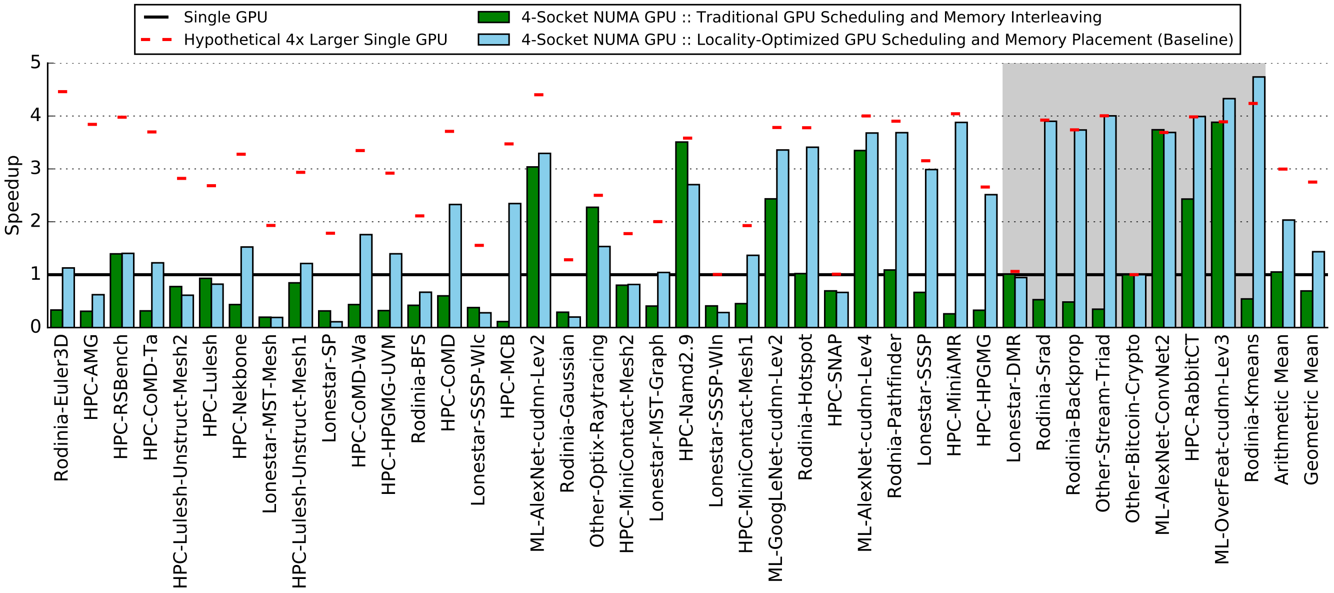


Figure 3: Performance of a 4-socket NUMA GPU relative to a single GPU and a hypothetical 4× larger (all resources scaled) single GPU. Applications shown in grey achieve greater than 99% of performance scaling with SW-only locality optimization.

data locality present in applications where contiguous CTAs also access contiguous memory regions [3, 7].

3.1 Performance Through Locality

Figure 3 shows the relative performance of a 4-socket NUMA GPU compared to a single GPU using the two possible CTA scheduling and memory placement strategies explained above. The green bars show the relative performance of traditional single-GPU scheduling and memory interleaving policies when adapted to a NUMA GPU. The light blue bars show the relative performance of using locality-optimized GPU scheduling and memory placement, consisting of contiguous block CTA scheduling and first touch page migration; after which pages are not dynamically moved between GPUs. The *Locality-Optimized* solution almost always outperforms the traditional GPU scheduling and memory interleaving. Without these runtime locality optimizations, a 4-socket NUMA GPU is not able to match the performance of a single GPU despite the large increase in hardware resources. Thus, using variants of prior proposals [3, 7], we now only consider this locality optimized GPU runtime for the remainder of the paper.

Despite the performance improvements that can come via locality-optimized software runtimes, many applications do not scale well on our proposed NUMA GPU system. To illustrate this effect, Figure 3 shows the speedup achievable by a hypothetical (unbuildable) 4× larger GPU with a red dash. This red dash represents an approximation of the maximum theoretical performance we expected from a perfectly architected (both HW and SW) NUMA GPU system. Figure 3 sorts the applications by the gap between relative performance of the Locality-Optimized NUMA GPU and hypothetical 4× larger GPU. We observe that on the right side of the graph some workloads (shown in the grey box) can achieve or surpass the maximum theoretical performance. In particular for the two far-most benchmarks on the right, the locality optimized solutions can outperform the hypothetical 4× larger GPU due higher cache hit rates because contiguous block scheduling is more cache friendly than traditional GPU scheduling.

However, the applications on the left side show a large gap between the Locality-Optimized NUMA design and theoretical performance. These are workloads in which either locality does not exist or the Locality-Optimized GPU runtime is not effective, resulting in large number of remote data accesses. Because our goal is to provide scalable performance for single-GPU optimized applications, the rest of the paper describes how to close this performance gap through microarchitectural innovation. To simplify later discussion, we choose to exclude benchmarks that achieve ≥99% of the theoretical performance with software-only locality optimizations. Still, we include all benchmarks in our final results to show the overall scalability achievable with NUMA-aware multi-socket GPUs.

3.2 Simulation Methodology

To evaluate the performance of future NUMA-aware multi-socket GPUs we use a proprietary, cycle-level, trace-driven simulator for single and multi-GPU systems. Our baseline GPU in both single GPU and multi-socket GPU configurations, approximates the latest NVIDIA Pascal architecture [43]. Each streaming multiprocessors (SM) is modeled as an in-order processor with multiple levels of cache hierarchy containing private, per-SM, L1 caches and a multi-banked, shared, L2 cache. Each GPU is backed by local on-package high bandwidth memory [23]. Our multi-socket GPU systems contains two to eight of these GPUs interconnected through a high bandwidth switch as shown in Figure 1. Table 1 provides a more detailed overview of the simulation parameters.

Parameter	Value(s)
Num of GPU sockets	4
Total number of SMs	64 per GPU socket
GPU Frequency	1 GHz
Max number of Warps	64 per SM
Warp Scheduler	Greedy then Round Robin
L1 Cache	Private, 128 KB per SM, 128 B lines, 4-way, Write-Through, GPU-side SW-based coherent
L2 Cache	Shared, Banked, 4 MB per socket, 128 B lines, 16-way, Write-Back, Mem-side non-coherent
GPU–GPU Interconnect	128 GBs per socket (64 GBs each direction) 8 lanes 8 B wide each per direction 128-cycle latency
DRAM Bandwidth	768 GBs per GPU socket
DRAM Latency	100 ns

Table 1: Simulation parameters for evaluation of single and multi-socket GPU systems.

Benchmark	Time-weighted Average CTAs	Memory Footprint (MB)
ML-GoogLeNet-cudnn-Lev2	6272	1205
ML-AlexNet-cudnn-Lev2	1250	832
ML-OverFeat-cudann-Lev3	1800	388
ML-AlexNet-cudnn-Lev4	1014	32
ML-AlexNet-ConvNet2	6075	97
Rodinia-Backprop	4096	160
Rodinia-Euler3D	1008	25
Rodinia-BFS	1954	38
Rodinia-Gaussian	2599	78
Rodinia-Hotspot	7396	64
Rodinia-Kmeans	3249	221
Rodnia-Pathfinder	4630	1570
Rodinia-Srad	16384	98
HPC-SNAP	200	744
HPC-Nekbone-Large	5583	294
HPC-MiniAMR	76033	2752
HPC-MiniContact-Mesh1	250	21
HPC-MiniContact-Mesh2	15423	257
HPC-Lulesh-Unstruct-Mesh1	435	19
HPC-Lulesh-Unstruct-Mesh2	4940	208
HPC-AMG	241549	3744
HPC-RSBench	7813	19
HPC-MCB	5001	162
HPC-NAMD2.9	3888	88
HPC-RabbitCT	131072	524
HPC-Lulesh	12202	578
HPC-CoMD	3588	319
HPC-CoMD-Wa	13691	393
HPC-CoMD-Ta	5724	394
HPC-HPGMG-UVM	10436	1975
HPC-HPGMG	10506	1571
Lonestar-SP	75	8
Lonestar-MST-Graph	770	86
Lonestar-MST-Mesh	895	75
Lonestar-SSSP-Wln	60	21
Lonestar-DMR	82	248
Lonestar-SSSP-Wlc	163	21
Lonestar-SSSP	1046	38
Other-Stream-Triad	699051	3146
Other-Optix-Raytracing	3072	87
Other-Bitcoin-Crypto	60	5898

Table 2: Time-weighted average number of thread blocks and application footprint.

GPU coherence protocols are not one-size fits all [49, 54, 63]. This work examines clusters of large discrete GPUs but smaller more tightly integrated GPU–CPU designs exist today as system on chips (SoC) [12, 33]. In these designs GPUs and CPUs can share a single memory space and last-level cache, necessitating a compatible GPU–CPU coherence protocol. However closely coupled CPU-GPU solutions are not likely to be ideal candidates for GPU-centric HPC workloads. Discrete GPUs each dedicate tens of billions of transistors to throughput computing, while integrated solutions dedicate only a fraction of the chip area. While discrete GPUs are also starting to integrate more closely with some CPU coherence protocols [1, 63], PCIe attached discrete GPUs (where integrated coherence is not possible) are likely to continue dominating the market, thanks to broad compatibility between CPU and GPU vendors.

This work examines the scalability of one such cache coherence protocol used by PCIe attached discrete GPUs. The protocol is optimized for simplicity and without need for hardware coherence support at any level of the cache hierarchy. SM-side L1 private caches achieve coherence through compiler inserted cache control (flush) operations and memory-side L2 caches, which do not require coherence support. While software-based coherence may seem heavy handed compared to fine grained MOESI-style hardware coherence, many GPU programming models (in addition to C++ 2011) are moving towards scoped synchronization where explicit software acquire and release operations must be used to enforce coherence. Without the use of these operations, coherence is not globally guaranteed and thus maintaining fine grain CPU-style MOESI coherence (via either directories or broadcast) may be an unnecessary burden.

We study the scalability of multi-socket NUMA GPUs using 41 workloads taken from a range of production codes based on the HPC CORAL benchmarks [28], graph applications from Lonestar [45], HPC applications from Rodinia [9], and several other in-house CUDA benchmarks. This set of workloads covers a wide spectrum of GPU applications used in machine learning, fluid dynamic, image manipulation, graph traversal, and scientific computing. Each of our benchmarks is hand selected to identify the region of interest deemed representative for each workload, which may be as small as a single kernel containing a tight inner loop or several thousand kernel invocations. We run each benchmark to completion for the determined region of interest. Table 2 provides both the memory footprint per application as well as the average number of active CTAs in the workload (weighted by the time spent on each kernel) to provide a representation of how many parallel thread blocks (CTAs) are generally available during workload execution.

4 ASYMMETRIC INTERCONNECTS

Figure 4(a) shows a switch connected GPU with symmetric and static link bandwidth assignment. Each link consists of equal numbers of uni-directional high-speed lanes in both directions, collectively comprising a symmetric bi-directional link. Traditional static-design time

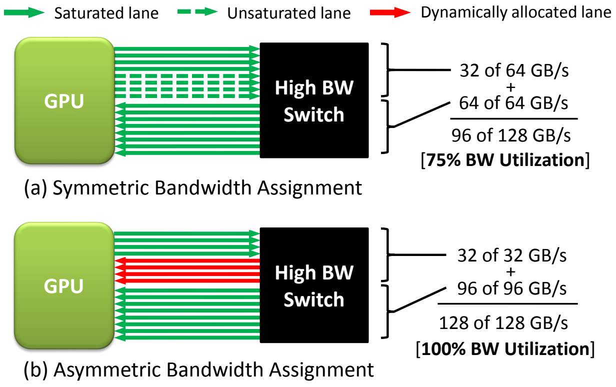


Figure 4: Example of dynamic link assignment to improve interconnect efficiency.

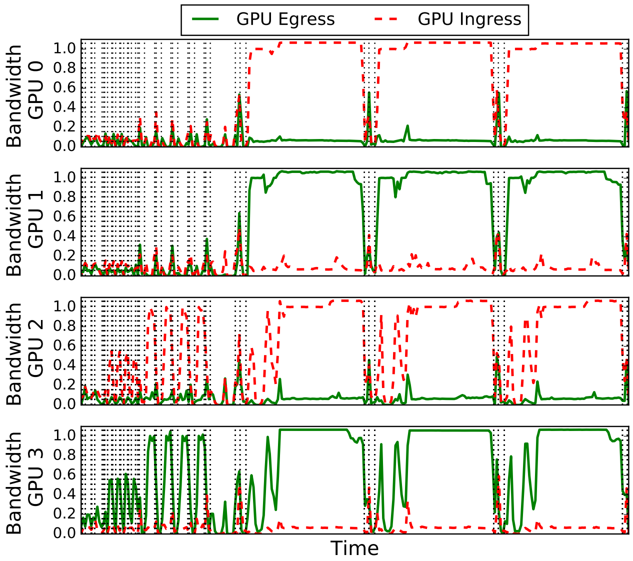


Figure 5: Normalized link bandwidth profile for HPC-HPGMG-UVM showing asymmetric link utilization between GPUs and within a GPU. Vertical black dotted lines indicate kernel launch events.

link capacity assignment is very common and has several advantages. For example, only one type of I/O circuitry (egress drivers or ingress receivers) along with only one type of control logic are implemented at each on-chip link interface. Moreover, the multi-socket switches result in simpler designs that can easily support a statically provisioned bandwidth requirements. On the other hand, multi-socket link bandwidth utilization can have a large influence on overall system performance. Static partitioning of bandwidth, when application needs are dynamic, can leave performance on the table. Because I/O bandwidth is a limited and expensive system resource, NUMA-aware interconnect designs must look for innovations that can keep wire and I/O utilization high.

In multi-socket NUMA GPU systems, we observe that many applications have different utilization of egress and ingress channels on both a per GPU-socket basis and during different phases of execution. For example, Figure 5 shows a link utilization snapshot over time for HPC-HPGMG-UVM benchmark running on a SW locality-optimized 4-socket NUMA GPU. Vertical dotted black lines represent kernel invocations that are split across the 4 GPU sockets. Several small kernels have negligible interconnect utilization. However, for the later larger kernels, GPU0 and GPU2 fully saturate their ingress links, while GPU1 and GPU3 fully saturate their egress links. At the same time GPU0 and GPU2, and GPU1 and GPU3 are underutilizing their egress and ingress links respectively.

In many workloads a common scenario has CTAs writing to the same memory range at the end of a kernel (i.e. parallel reductions, data gathering). For CTAs running on one of the sockets, GPU0 for example, these memory references are local and do not produce any traffic on the inter-socket interconnections. However CTAs dispatched to other GPUs must issue remote memory writes, saturating their egress links while ingress links remain underutilized, but causing ingress traffic on GPU0. Such communication patterns typically utilize only 50% of available interconnect bandwidth. In these cases, dynamically increasing the number of ingress lanes for GPU0 (by reversing the direction of egress lanes) and switching the direction of ingress lanes for GPUs 1–3, can substantially improve the achievable interconnect bandwidth. Motivated by these findings, we propose to dynamically control multi-socket link bandwidth assignments on a per-GPU basis, resulting in dynamic asymmetric link capacity assignments as shown in Figure 4(b).

To evaluate this proposal, we model point-to-point links containing multiple lanes, similar to PCIe [47] or NVLink [43]. In these links, 8 lanes with 8 GBs capacity per lane yield an aggregate bandwidth of 64 GBs in each direction. We propose replacing uni-directional lanes with bi-directional lanes to which we apply an adaptive link bandwidth allocation mechanism that works as following. For each link in the system, at kernel launch the links are always reconfigured to contain symmetric link bandwidth with 8 lanes per direction. During kernel execution the link load balancer periodically samples the saturation status of each link. If the lanes in one direction are not saturated, while the lanes in the opposite direction are 99% saturated, the link load balancer reconfigures and reverses the direction of one of the unsaturated lanes after quiescing all packets on that lane.

This sample and reconfigure process stops only when directional utilization is not oversubscribed or all but one lane is configured in a single direction. If both ingress and egress links are saturated and in an asymmetric configuration, links are then reconfigured back toward a symmetric configuration to encourage global bandwidth equalization. While this process may sound complex, the circuitry for dynamically turning high speed single ended links around in just tens of cycles or less is already in use by modern high bandwidth memory interfaces, such as GDDR, where the same set of wires is used for both memory reads and writes [16]. In high speed signaling implementations, necessary phase–delay lock loop resynchronization can occur while data is inflight; eliminating the need to idle the link during this long latency (microseconds) operation if upcoming link turn operations can be sufficiently projected ahead of time, such as on a fixed interval.

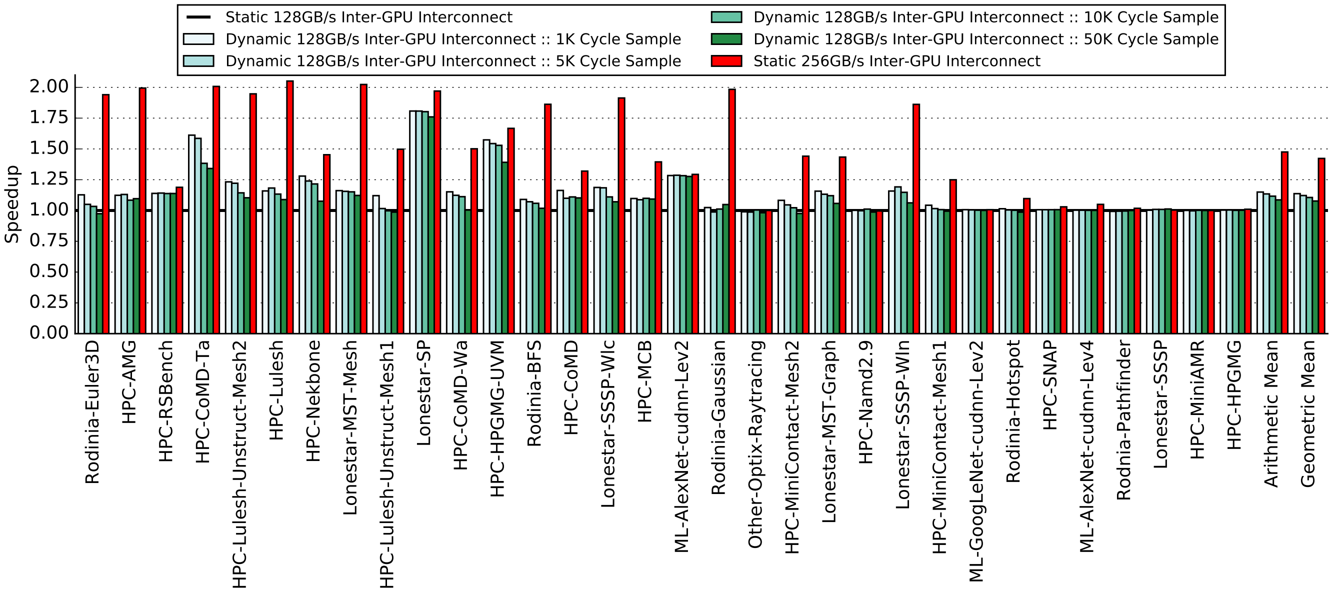


Figure 6: Relative speedup of the dynamic link adaptivity compared to the baseline architecture by varying sample time and assuming switch time of 100 cycles. In red, we also provide speedup achievable by doubling link bandwidth.

4.1 Results and Discussion

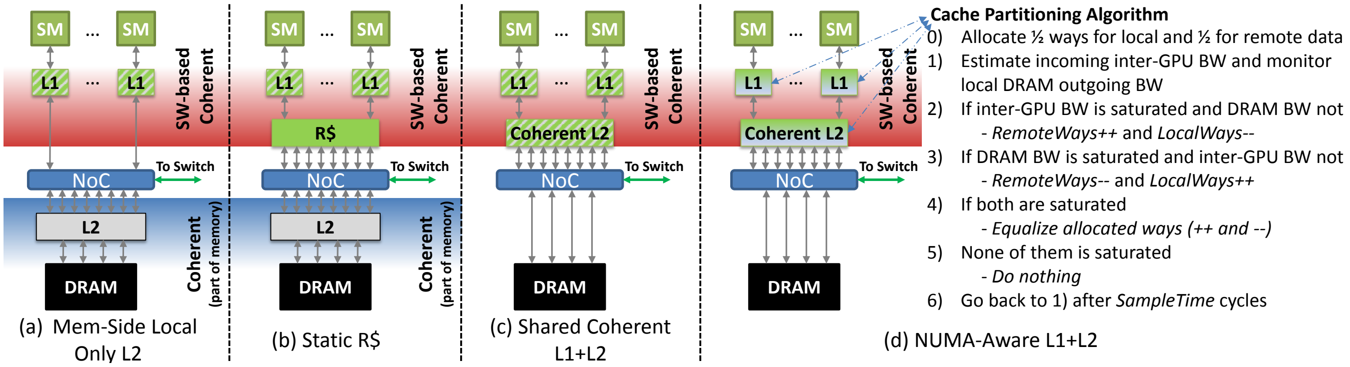
Two important parameters affect the performance of our proposed mechanism: (i) `SampleTime` – the frequency at which the scheme samples for a possible reconfiguration and (ii) `SwitchTime` – the cost of turning the direction of an individual lane. Figure 6 shows the performance improvement, compared to our locality-optimized GPU by exploring different values of the `SampleTime` indicated by green bars and assuming a `SwitchTime` of 100 cycles. The red bars in Figure 6 provide an upper-bound of performance speedups when doubling the available interconnect bandwidth to 256 GBs. For workloads on the right of the figure, doubling the link bandwidth has little effect, indicating that a dynamic link policy will also show little improvement due to low GPU–GPU interconnect bandwidth demand. On the left side of the figure, for some applications when improved interconnect bandwidth has a large effect, dynamic lane switching can improve application performance by as much as 80%. For some benchmarks like `Rodinia-Euler-3D`, `HPC-AMG`, and `HPC-Lulesh`, doubling the link bandwidth provides 2× speedup, while our proposed dynamic link assignment mechanism is not able to significantly improve performance. These workloads saturate both link directions, so there is no opportunity to provide additional bandwidth by turning links around.

Using a moderate 5K cycle sample time, the dynamic link policy can improve performance by 14% on average over static bandwidth partitioning. If the link load balancer samples too infrequently, application dynamics can be missed and performance improvement is reduced. However if the link is reconfigured too frequently, bandwidth is lost due to the overhead of turning the link. While we have assumed a pessimistic link turn time of 100 cycles, we performed sensitivity studies that show even if link turn time were increased to 500 cycles, our dynamic policy loses less than 2% in performance. At the same time, using a faster lane switch (10 cycles) does not significantly improve the performance over a 100 cycle link turn time. The link turnaround times of modern high-speed on-board links such as GDDR5 [16] are about 8 ns with both link and internal DRAM turn-around latency, which is less than 10 cycles at 1 GHz.

Our results demonstrate that asymmetric link bandwidth allocation can be very attractive when inter-socket interconnect bandwidth is constrained by the number of on-PCB wires (and thus total link bandwidth). The primary drawback of this solution is that both types of interface circuitry (TX and RX) and logic must be implemented for each lane in both the GPU and switch interfaces. We conducted an analysis of the potential cost of doubling the amount of I/O circuitry and logic based on a proprietary state of the art GPU I/O implementation. Our results show that doubling this interface area increases total GPU area by less than 1% while yielding a 12% improvement in average interconnect bandwidth and a 14% application performance improvement. One additional caveat worth noting is that the proposed asymmetric link mechanism optimizes link bandwidth in a given direction for each individual link, while the total switch bandwidth remains constant.

5 NUMA-AWARE CACHE MANAGEMENT

Section 4 showed that inter-socket bandwidth is an important factor in achieving scalable NUMA GPU performance. Unfortunately, because either the outgoing or incoming links must be underutilized for us to reallocate that bandwidth to the saturated link, if both incoming and outgoing links are saturated, dynamic link rebalancing yields minimal gains. To improve performance in situations where dynamic link balancing is ineffective, system designers can either increase link bandwidth, which is very expensive, or decrease the amount of traffic that crosses the low bandwidth communication channels. To



(a) Mem-Side Local Only L2

(b) Static R$

(c) Shared Coherent L1+L2

(d) NUMA-Aware L1+L2

Cache Partitioning Algorithm

0) Allocate ½ ways for local and ½ for remote data
1) Estimate incoming inter-GPU BW and monitor local DRAM outgoing BW
2) If inter-GPU BW is saturated and DRAM BW not
   - *RemoteWays++* and *LocalWays--*
3) If DRAM BW is saturated and inter-GPU BW not
   - *RemoteWays--* and *LocalWays++*
4) If both are saturated
   - *Equalize allocated ways (++ and --)*
5) None of them is saturated
   - *Do nothing*
6) Go back to 1) after *SampleTime* cycles

Figure 7: Potential L2 cache organizations to balance capacity between remote and local NUMA memory systems.

decrease off-chip memory traffic, architects typically turn to caches to capture locality.

GPU cache hierarchies differ from traditional CPU hierarchies as they typically do not implement strong hardware coherence protocols [55]. They also differ from CPU protocols in that caches may be both processor side (where some form of coherence is typically necessary) or they may be memory side (where coherence is not necessary). As described in Table 1 and Figure 7(a), a GPU today is typically composed of relatively large SW managed coherent L1 caches located close to the SMs, while a relatively small, distributed, non-coherent memory side L2 cache resides close to the memory controllers. This organization works well for GPUs because their SIMT processor designs often allow for significant coalescing of requests to the same cache line, so having large L1 caches reduces the need for global crossbar bandwidth. The memory-side L2 caches do not need to participate in the coherence protocol, which reduces a system complexity.

5.1 Design Considerations

In NUMA designs, remote memory references occurring across low bandwidth NUMA interconnections results in poor performance, as shown in Figure 3. Similarly, in NUMA GPUs utilizing traditional memory-side L2 caches (that depend on fine grained memory interleaving for load balancing) is a bad decision. Because memory-side caches only cache accesses that originate in their local memory, they cannot cache memory from other NUMA zones and thus can not reduce NUMA interconnect traffic. Previous work has proposed that GPU L2 cache capacity should be split between memory-side caches and a new processor-side L1.5 cache that is an extension of the GPU L1 caches [3] to enable caching of remote data, shown in Figure 7(b). By balancing L2 capacity between memory side and remote caches (R$), this design limits the need for extending expensive coherence operations (invalidations) into the entire L2 cache while still minimizing crossbar or interconnect bandwidth.

Flexibility: Designs that statically allocate cache capacity to local memory and remote memory, in any balance, may achieve reasonable performance in specific instances but they lack flexibility. Much like application phasing was shown to affect NUMA bandwidth consumption the ability to dynamically share cache capacity between local and remote memory has the potential to improve performance under several situations. First, when application phasing results in some GPU-sockets primarily accessing data locally while others are accessing data remotely, a fix partitioning of cache capacity is guaranteed to be sub-optimal. Second, while we show that most applications will be able to completely fill large NUMA GPUs, this may not always be the case. GPUs within the data center are being virtualized and there is continuous work to improve concurrent execution of multiple kernels and processes within a single GPU [15, 32, 46, 50]. If a large NUMA GPU is sub-partitioned, it is intuitive that system software attempt to partition it along the NUMA boundaries (even within a single GPU-socket) to improve the locality of small GPU kernels. To effectively capture locality in these situation, NUMA-aware GPUs need to be able to dynamically re-purpose cache capacity at runtime, rather than be statically partitioned at design time.

Coherence: To-date, discrete GPUs have not moved their memside caches to processor side because the overhead of cache invalidation (due to coherence) is an unnecessary performance penalty. For a single socket GPU with a uniform memory system, there is little performance advantage to implementing L2 caches as processor side caches. Still, in a multi-socket NUMA design, the performance tax of extending coherence into L2 caches is offset by the fact that remote memory accesses can now be cached locally and may be justified. Figure 7(c) shows a configuration with a coherent L2 cache where remote and local data contend for L2 capacity as extensions of the L1 caches, implementing identical coherence policy.

Dynamic Partitioning: Building upon coherent GPU L2 caches, we posit that while conceptually simple, allowing both remote and local memory accesses to contend for cache capacity (in both the L1 and L2 caches) in a NUMA system is flawed. In UMA systems it is well know that performance is maximized by optimizing for cache hit rate, thus minimizing off-chip memory system bandwidth. However, in NUMA systems, *not all cache misses have the same relative cost performance impact.* A cache miss to a local memory address has a smaller cost (in both terms of latency and bandwidth) than a cache miss to a remote memory address. Thus, it should be beneficial to dynamically skew cache allocation to preference caching remote memory over local data when it is determined the system is bottle-necked on NUMA bandwidth.

To minimize inter-GPU bandwidth in multi-socket GPU systems we propose a NUMA-aware cache partitioning algorithm, with cache organization and brief summary shown in Figure 7(d). Similar to

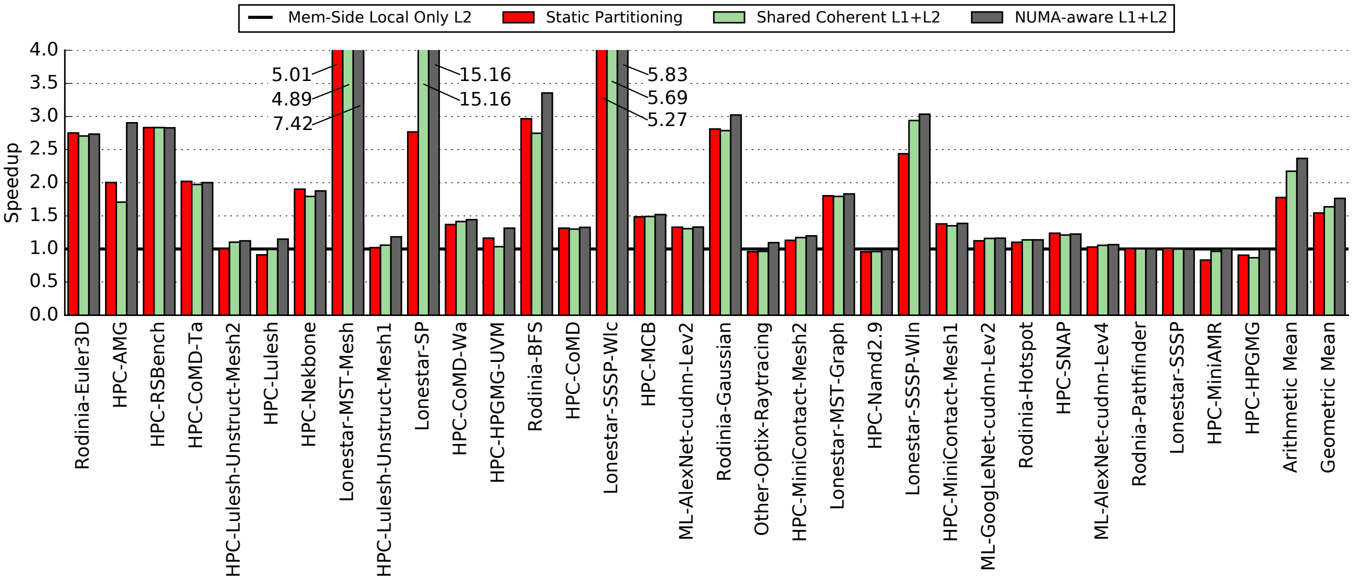


Figure 8: Performance of 4-socket NUMA-aware cache partitioning, compared to memory-side L2 and static partitioning.

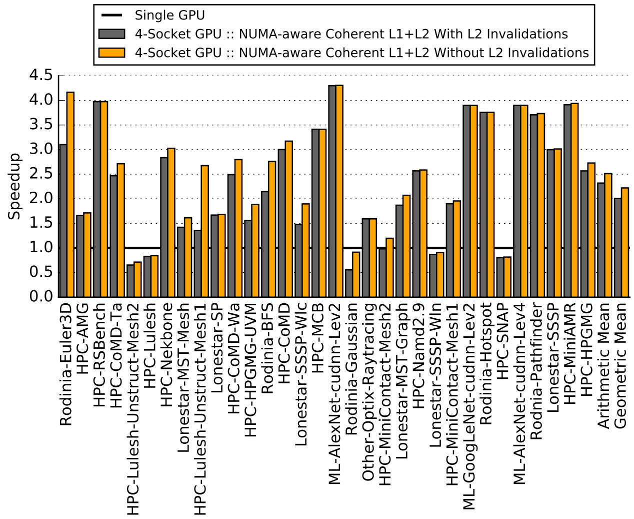


Figure 9: Performance overhead of extending current GPU software based coherence into the GPU L2 caches.

our interconnect balancing algorithm, at initial kernel launch (after GPU caches have been flushed for coherence purposes) we allocate one half of the cache ways for local memory and the remaining ways for remote data (Step ⓪). After executing for a 5K cycles period, we sample the average bandwidth utilization on local memory and estimate the GPU-socket's incoming read request rate by looking at the outgoing request rate multiplied by the response packet size. By using the outgoing request rate to estimate the incoming bandwidth, we avoid situations where incoming writes may saturate our link bandwidth falsely indicating we should preference remote data caching. Projected link utilization above 99% is considered to be bandwidth saturated (Step ①). In cases where the interconnect bandwidth is saturated but local memory bandwidth is not, the partitioning algorithm attempts to reduce remote memory traffic by re-assigning one way from the group of local ways to the remote ways grouping (Step ②). Similarly, if the local memory BW is saturated and inter-GPU bandwidth is not, the policy re-allocates one way from the remote group, and allocates it to the group of local ways (Step ③). To minimize the impact on cache design, all ways are consulted on look up, allowing lazy eviction of data when the way partitioning changes. In case where both the interconnect and local memory bandwidth are saturated, our policy gradually equalizes the number of ways assigned for remote and local cache lines (Step ④). Finally, if neither of the links are currently saturated, the policy takes no action (Step ⑤). To prevent cache starvation of either local or remote memory (which causes memory latency dramatically increase and a subsequent drop in performance), we always require at least one way in all caches to be allocated to either remote of local memory.

5.2 Results

Figure 8 compares the performance of 4 different cache configurations in our 4-socket NUMA GPU. Our baseline is a traditional GPU with memory-side local-only write-back L2 caches. To compare against prior work [3] we provide a 50–50 static partitioning where the L2 cache budget is split between the GPU-side coherent remote cache which contains only remote data, and the memory side L2 which contains only local data. In our 4-socket NUMA GPU static partitioning improves performance by 54% on average, although for some benchmarks, it hurts the performance by as much as 10% for workloads that have negligible inter-socket memory traffic. We also show the results for GPU-side coherent L1 and L2 caches where both local and remote data contend capacity. On average, this solution

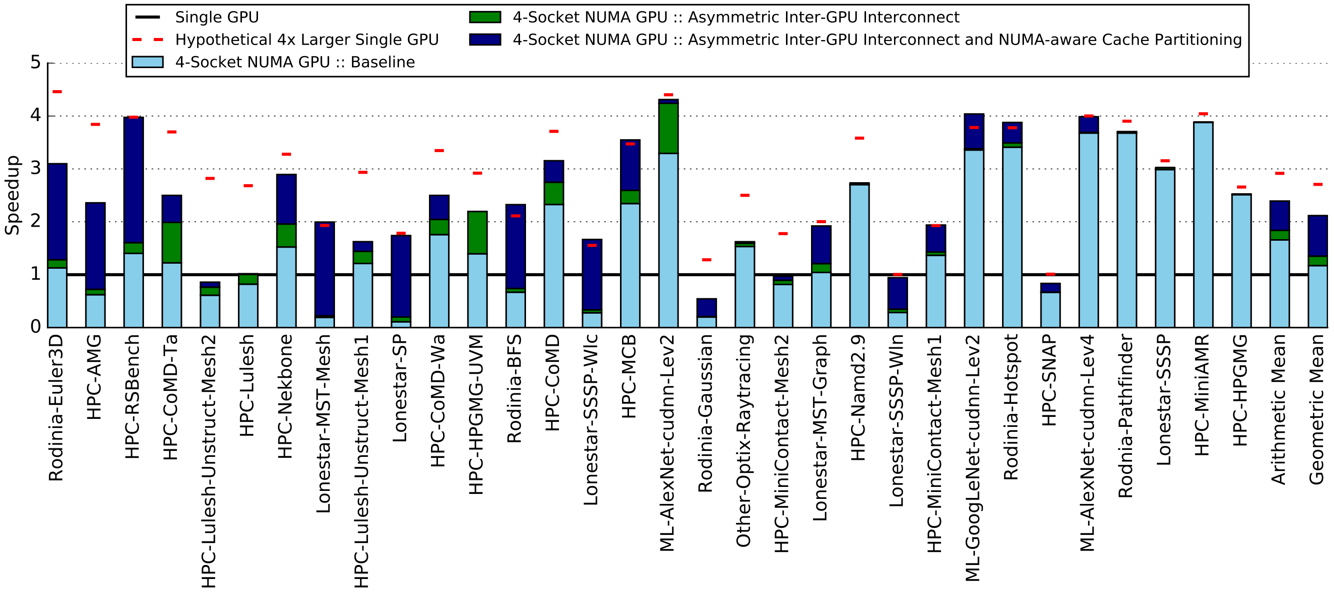


Figure 10: Final NUMA-aware GPU performance compared to a single GPU and 4× larger single GPU with scaled resources.

outperforms static cache partitioning significantly despite incurring additional flushing overhead due to cache coherence.

Finally, our proposed NUMA-aware cache partitioning policy is shown in dark grey. Due to its ability to dynamically adapt the capacity of both L2 and L1 to optimize performance when backed by NUMA memory, it is the highest performing cache configuration. By examining simulation results we find that for workloads on the left side of Figure 8 which fully saturate the inter-GPU bandwidth, NUMA-aware dynamic policy configures the L1 and L2 caches to be primarily used as remote caches. However, workloads on the right side of the figure tend to have good GPU-socket memory locality, and thus prefer L1 and L2 caches store primarily local data. NUMA-aware cache partitioning is able to flexibly adapt to varying memory access profiles and can improve average NUMA GPU performance 76% compared to traditional memory side L2 caches, and 22% compared to previously proposed static cache partitioning despite incurring additional coherence overhead.

When extending SW-controlled coherence into the GPU L2 caches, L1 coherence operations must be extended into the GPU L2 caches. Using bulk software invalidation to maintain coherence is simple to implement but is a performance penalty when falsely evicting not required data. The overhead of this invalidation is dependent on both the frequency of the invalidations as well as aggregate cache capacity invalidated. Extending the L1 invalidation protocol into the shared L2, and then across multiple GPUs, increases the capacity affected and frequency of the invalidation events.

To understand the impact of these invalidations, we evaluate hypothetical L2 caches which can ignore the cache invalidation events; thus representing the upper limit on performance (no coherence evictions ever occur) that could be achieved by using a finer granularity HW-coherence protocol. Figure 9 shows the impact these invalidation operations have on application performance. While significant for some applications, on average SW-based cache invalidations overheads are only 10%, even when extended across all GPU-socket L2 caches. So while fine grain HW coherence protocols may improve performance, the magnitude of their improvement must be weighted against their hardware implementation complexity. While in the studies above we assumed a write-back policy in L2 caches, as a sensitivity study we also evaluated the effect of using a write-through cache policy to mirror the write-through L1 cache policy. Our findings indicate that write-back L2 outperforms write-through L2 by 9% on average in our NUMA-GPU design due to the decrease in total inter-GPU write bandwidth.

6 DISCUSSION

Combined Improvement: Sections 4 and 5 provide two techniques aimed at more efficiently utilizing scarce NUMA bandwidth within future NUMA GPU systems. The proposed methods for dynamic interconnect balancing and NUMA-aware caching are orthogonal and can be applied in isolation or combination. Dynamic interconnect balancing has an implementation simplicity advantage in that the system level changes to enable this feature are isolated from the larger GPU design. Conversely, enabling NUMA-aware GPU caching based on interconnect utilization requires changes to both the physical cache architecture and the GPU coherence protocol.

Because these two features target the same problem, when employed together their effects are not strictly additive. Figure 10 shows the overall improvement NUMA-aware GPUs can achieve when applying both techniques in parallel. For benchmarks such as `CoMD`, these features contribute nearly equally to the overall improvement, but for others such as `ML-AlexNet-cudnn-Lev2` or `HPC-MST-Mesh1`, interconnect improvements or caching are the primary contributor respectively. On average, we observe that when combined we see 2.1× improvement over a single GPU and 80% over the baseline

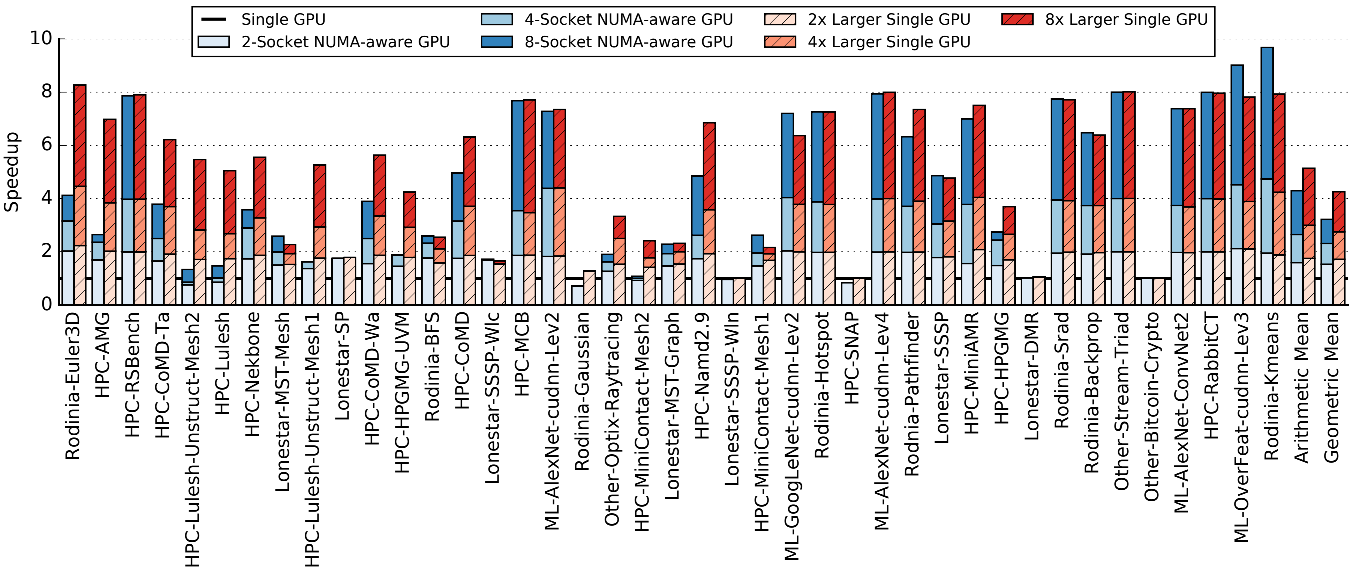


Figure 11: 1–8 socket NUMA-aware GPU scalability compared to hypothetical larger single GPU with scaled resources.

software locality optimized 4-socket NUMA GPU using memory side L2 caches; best performance is clearly obtained when applying both features in unison.

Scalability: Ultimately, for vendors to produce multi-socket NUMA GPUs they must achieve high enough parallel efficiency to justify their design. To understand the scalability of our approach Figure 11 shows the performance of a NUMA-aware multi-socket GPU compared to a single GPU, when scaled across 2, 4, and 8 sockets respectively. On average a 2 socket NUMA GPU achieves 1.5× speedup, while 4 sockets and 8 sockets achieve 2.3× and 3.2× speedups respectively. Depending on perspective these speedups may look attractive or lackluster; particularly when per-benchmark variance is included. However, the scalability of NUMA GPUs is not solely dependent on just NUMA GPU microarchitecture. We observe that for some applications, even if the application was run on larger hypothetical single GPUs, performance would scale similarly. This may be due to a variety of reasons beyond NUMA effects, including number of CTAs available, frequency of global synchronization, and other factors. Comparing our NUMA-aware GPU implementation to the scaling that applications could achieve on a hypothetical large single GPU, we see that NUMA-GPUs can achieve 89%, 84%, and 76% the efficiency of a hypothetical single large GPU in 2, 4, and 8 socket configurations respectively. This high efficiency factor indicates that our design is able to largely eliminate the NUMA penalty in future multi-socket GPU designs.

Multi-Tenancy on Large GPUs: In this work we have shown that many workloads today have the ability to saturate (with sufficient parallel work) a GPU that is at least 8× larger than today's GPUs. With deep-data becoming commonplace across many computing paradigms, we believe that the trend of having enough parallel thread blocks to saturate large single GPUs will continue into the foreseeable future. However when GPUs become larger at the expense of having multiple addressable GPUs within the system, questions related to GPU provisioning arise. Applications that cannot saturate large GPUs will leave resources underutilized and concurrently will have to multiplex across the GPU cooperatively in time, both undesirable outcomes.

While not the focus of this work, there is significant effort in both industry and academia to support finer grain sharing of GPUs through either shared SM execution [57], spatial multiplexing of a GPU [46], or through improved time division multiplexing with GPU pre-emptability [32]. To support large GPU utilization any of these solutions could be applied to a multi-socket GPU in the cases where applications may not completely fill a larger GPU. Alternatively, with additional GPU runtime work multi-socket GPU designs could also be dynamically partitioned with a granularity of 1–N logical GPUs being exposed to the programmer, providing yet another level of flexibility to improve utilization.

Power Implications: As discussed earlier, arbitrarily large monolithic single GPUs are unfeasible, so multi-GPU systems connected by high-speed links and switches may become attractive solutions for continuing GPU performance scaling. However, onboard high-speed links and switches require additional power. We estimated the link overhead by assuming 10 pJb of on board interconnect energy for combined links and switch (extrapolated from publicly available information for cabinet level Mellanox switches and links [58, 59]). Using this estimate we calculate an average (Geo-Mean) 30 W of communication power for the baseline architecture composed of 4 GPUs, and 14 W after our NUMA-aware optimizations are applied. Some applications such as `Rodinia-Euler3D, HPC-Lulesh, HPC-AMG, HPC-Lulesh-Unstruct-Mesh2` are communication intensive, resulting in ≈130 W of power consumption after our optimizations are considered. Assuming a typical TDP of 250 W per GPU module, in a 4–GPU system, the extra power due to the communication represents a 5% overhead across the full range of 41 evaluated benchmarks. While this power tax is not trivial, without alternative methods for building large GPUs, interconnect power will likely become a large portion of the total GPU power budget.

7 RELATED WORK

Multi-GPU programming is commonly used for scaling GPU performance via integration of multiple GPUs at the system level [26, 35, 43, 61] for a rapidly growing pool of applications [27, 28, 34, 53]. Similarly, multi-socket and multi-node CPU installations have been employed and studied in context of high performance computing and datacenter applications [11, 18–20]. Multi-GPU programming requires explicit design for multiple GPUs using SW APIs such as Peer-2-Peer access [36] or a combination of MPI and CUDA [42]. These extensions require unique programming experience and significant SW effort while adapting a traditional GPU application to a multi-GPU system. In this paper we execute single GPU applications on a NUMA multi-GPU system as if it was a single larger GPU via hardware innovations and extensions to the driver software stack; providing programmer and OS transparent execution, similarly to approaches proposed in the past [4, 7, 30].

Modern multi-socket CPU and GPU systems leverage advanced interconnect technologies such as NVLink, QPI and Infinity [2, 21, 35]. These modern fabrics utilize high speed serial signalling technologies over unidirectional lanes collectively comprising full-duplex links. Link capacity is statically allocated at design time and usually is symmetric in nature. Asymmetric network on chips and NUMA interconnects have been previously investigated and deployed [48, 62]. In this paper we propose to dynamically re-allocate available link bandwidth resources by using same system wiring resources and on-chip I/O interfaces, while implementing both receiver and transmitter driver circuitry on each lane. This approach resembles previously proposed tristate bi-directional bus technologies [56] or former technologies such as the Intel front-side bus [10], albeit with just two bus clients. However our proposal leverages fast singled ended signalling while allowing a dynamically controlled asymmetric bandwidth allocation via on-the-fly reconfiguration of the individual lane direction within a link.

Static and dynamic cache partitioning techniques were widely explored in the context of CPU caches and QoS [8, 14, 22, 51, 52] For example, Rafique et. al [52] proposed architectural support for shared cache management with quota-based approach. Qureshi et. al [51] proposed to partition cache space between applications. Jaleel et. al [22] improved on this by proposing adaptive insertion policies. Recently, cache monitoring and allocation technologies were added to Intel Xeon processors, targeted for QoS enforcement via dynamic repartitioning of on-chip CPU cache resources [14] between applications. Efficient cache partitioning in the GPU has been explored in context of L1 caches [31] to improve application throughput. While dynamic cache partitioning has been widely used for QoS and L1 utilization, to the best of our knowledge it has never been used to try to optimize performance when caches are backed by NUMA memory systems.

8 CONCLUSIONS

With transistors growth slowing and multi-GPU programming requiring re-architecting of GPU applications, the future of scalable single GPU performance is in question. We propose that much like CPU designs have done in the past, the natural progression for continuous performance scalability of traditional GPU workloads is to move from a single to multi-socket NUMA design. In this work we show that applying NUMA scheduling and memory placement policies inherited from the CPU world is not sufficient to achieve good performance scalability. We show that future GPU designs will need to become NUMA-aware both in their interconnect management and within their caching subsystems to overcome the inherent performance penalty that NUMA memory systems introduce. By leveraging software policies that preserve data locality and hardware policies that can dynamically adapt to application phases, our proposed NUMA-aware multi-socket GPU is able to outperform current GPU designs by 1.5×, 2.3×, and 3.2×, while achieving 89%, 84%, and 76% of theoretical application scalability in 2, 4, and 8 GPU sockets respectively. Our results indicate that the challenges of designing a multi-socket NUMA GPU can be solved through a combination of runtime and architectural optimization, making NUMA-aware GPUs a promising technology for scaling GPU performance beyond a single socket.

ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers and Steve Keckler for their help in improving this paper. The first author is supported by the Ministry of Economy and Competitiveness of Spain (TIN2012-34557, TIN2015-65316-P, and BES-2013-063925).

REFERENCES

[1] Neha Agarwal, David Nellans, Eiman Ebrahimi, Thomas F Wenisch, John Danskin, and Stephen W Keckler. 2016. Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.

[2] AMD Inc. 2017. AMD's Infinity Fabric Detailed. http://wccftech.com/amds-infinity-fabric-detailed/. (2017). [Online; accessed 2017-04-04].

[3] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *International Symposium on Computer Architecture (ISCA)*.

[4] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM.

[5] P. Bright. 2016. Moore's Law Really is Dead This Time. http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time. (2016). [Online; accessed 2017-04-04].

[6] Broadcom. 2017. PCI Express Switches. https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches/. (2017). [Online; accessed 2017-07-10].

[7] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B. Jablin, Nacho Navarro, and Wen-mei W. Hwu. 2015. Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*.

[8] J. Chang and G. S. Sohi. 2007. Cooperative Cache Partitioning for Chip Multiprocessors. *Proceedings of ISC* (June 2007).

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization (IISWC)*.

[10] INTEL Corporation. 2004. Intel Xeon Processor with 533 MHz Front Side Bus at 2 GHz to 3.20 GHz. http://download.intel.com/support/processors/xeon/sb/25213506.pdf. (2004). [Online; accessed 2017-04-04].

[11] INTEL Corporation. 2007. The Xeon X5365. http://ark.intel.com/products/30702/Intel-Xeon-Processor-X5365-8M-Cache-3_00-GHz-1333-MHz-FSB. (2007). [Online; accessed 2016-08-19].

[12] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* (2017).

[13] J. L. Hennessy and D. A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Elsevier.

[14] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. 2016. Cache QoS: From Concept to Reality in the Intel Xeon Processor E5-2600 v3 Product Family. *HPCA* (2016).

[15] HSA Fondation. 2016. HSA Platform System Architecture Specification 1.1. http://www.hsafoundation.com/?ddownload=5114. (2016). [Online; accessed 2017-13-06].

[16] SK Hynix. 2009. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Datasheet Revision 1.0. https://www.skhynix.com/eolproducts.view.do?pronm=GDDR5+SDRAM&srnm=H5GQ1H24AFR&rk=26&rc=graphics. (2009). [Online; accessed 2017-04-04].

[17] HyperTransport Consortium. 2010. HyperTransport 3.1 Specification. http://www.hypertransport.org/ht-3-1-link-spec. (2010). [Online; accessed 2017-04-04].

[18] IBM. 2011. IBM zEnterprise 196 Technical Guide. http://www.redbooks.ibm.com/redbooks/pdfs/sg247833.pdf. (2011). [Online; accessed 2017-04-04].

[19] IBM. 2012. IBM Power Systems Deep Dive. http://www-05.ibm.com/cz/events/febannouncement2012/pdf/power_architecture.pdf. (2012). [Online; accessed 2017-04-04].

[20] AMD Inc. 2012. AMD Server Solutions Playbook. http://www.amd.com/Documents/AMD_Opteron_ServerPlaybook.pdf. (2012). [Online; accessed 2017-04-04].

[21] INTEL Corporation. 2009. 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 2017-04-04].

[22] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, Jr S. Steely, and J. Emer. 2008. Adaptive Insertion Policies for Managing Shared Caches. *Proceedings of MICRO* (Oct 2008).

[23] JEDEC. 2015. High Bandwidth Memory(HBM) DRAM - JESD235. http://www.jedec.org/standards-documents/results/jesd235. (2015). [Online; accessed 2017-04-04].

[24] KHRONOS GROUP. 2016. OpenCL 2.2 API Specification (Provisional). https://www.khronos.org/opencl/. (2016). [Online; accessed 2017-04-04].

[25] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. *ACM SIGPLAN Notices* (2011).

[26] Oak Ridge National Laboratory. 2013. Titan : The World's #1 Open Science Super Computer. https://www.olcf.ornl.gov/titan/. (2013). [Online; accessed 2017-04-04].

[27] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. http://arxiv.org/abs/1509.09308. (2015). [Online; accessed 2017-04-04].

[28] Lawerence Livermore National Laboratory. 2014. CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks/. (2014). [Online; accessed 2017-04-04].

[29] Lawerence Livermore National Laboratory. 2016. CORAL/Sierra. https://asc.llnl.gov/coral-info. (2016). [Online; accessed 2017-04-04].

[30] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2013. Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE.

[31] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Keckler. 2015. Priority-Based Cache Allocation in Throughput Processors. *HPCA* (2015).

[32] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press.

[33] Benjamin Munger, David Akeson, Srikanth Arekapudi, Tom Burd, Harry R Fair, Jim Farrell, Dave Johnson, Guhan Krishnan, Hugh McIntyre, Edward McLellan, et al. 2016. Carrizo: A High Performance, Energy Efficient 28 nm APU. *Journal of Solid-State Circuits* (2016).

[34] NVIDIA. [n. d.]. NVIDIA cuDNN, GPU Accelerated Deep Learning. https://developer.nvidia.com/cudnn. ([n. d.]). [Online; accessed 2017-04-04].

[35] NVIDIA. [n. d.]. The World's First AI Supercomputer in a Box. http://www.nvidia.com/object/deep-learning-system.html. ([n. d.]). [Online; accessed 2017-04-04].

[36] NVIDIA. 2011. Multi-GPU Programming. http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf. (2011). [Online; accessed 2017-04-04].

[37] NVIDIA. 2013. Unified Memory in CUDA 6. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. (2013). [Online; accessed 2017-04-04].

[38] NVIDIA. 2014. Compute Unified Device Architecture. http://www.nvidia.com/object/cuda_home_new.html. (2014). [Online; accessed 2017-04-04].

[39] NVIDIA. 2014. 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-theway-to-exascale-computing. (2014). [Online; accessed 2017-04-04].

[40] NVIDIA. 2015. CUDA C Programming Guild v7.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (2015). [Online; accessed 2017-04-04].

[41] NVIDIA. 2016. Inside Pascal: NVIDIA's Newest Computing Platform. https://devblogs.nvidia.com/parallelforall/inside-pascal. (2016). [Online; accessed 2017-04-04].

[42] NVIDIA. 2016. MPI Solutions for GPUs. https://developer.nvidia.com/mpi-solutions-gpus. (2016). [Online; accessed 2017-04-04].

[43] NVIDIA. 2016. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. (2016). [Online; accessed 2017-04-04].

[44] NVIDIA. 2017. Scalable Link Interconnect. http://www.nvidia.in/object/sli-technology-overview-in.html. (2017). [Online; accessed 2017-07-10].

[45] M. A. O'Neil and M. Burtscher. 2014. Microarchitectural Performance Characterization of Irregular GPU Kernels. In *International Symposium on Workload Characterization (IISWC)*.

[46] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. *ACM SIGARCH Computer Architecture News* (2015).

[47] PCI-SIG. 2015. PCI Express Base Specification Revision 3.1a. https://members.pcisig.com/wg/PCI-SIG/document/download/8257. (2015). [Online; accessed 2017-07-10].

[48] Stephen Phillips. 2014. M7: Next Generation SPARC. In *Hot Chips 26 Symposium (HCS)*. IEEE.

[49] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous System Coherence for Integrated CPU-GPU Systems. In *International Symposium on Microarchitecture (MICRO)*.

[50] Sooraj Puthoor, Ashwin M Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M Beckmann, and Gregory Rodgers. 2016. Implementing Directed Acyclic Graphs With the Heterogeneous System Architecture. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM.

[51] M. Qureshi and Y. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *Proceedings of MICRO* (Dec 2006).

[52] N. Rafique, W.T. Lim, and M. Thottethodi. 2006. Architectural Support for OS-driven CMP Cache Management. *Proceedings of PACT* (Sep 2006).

[53] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. http://arxiv.org/abs/1409.1556. (2014). [Online; accessed 2017-04-04].

[54] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2015. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM.

[55] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. 2013. Cache Coherence for GPU Architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.

[56] John R. Spence and Michael M. Yamamura. 1985. Clocked Tri-State Driver Circuit. https://www.google.com/patents/US4504745. (1985).

[57] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling Preemptive Multiprogramming on GPUs. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[58] Mellanox Technologies. 2015. Switch-IB 2 EDR Switch Silicon - World's First Smart Switch. http://www.mellanox.com/related-docs/prod_silicon/PB_SwitchIB2_EDR_Switch_Silicon.pdf. (2015). [Online; accessed 2017-04-04].

[59] Mellanox Technologies. 2016. ConnectX-4 VPI Single and Dual Port QSFP28 Adapter Card User Manual. http://www.mellanox.com/related-docs/user_manuals/ConnectX-4_VPI_Single_and_Dual_QSFP28_Port_Adapter_Card_User_Manual.pdf. (2016). [Online; accessed 2017-04-04].

[60] Jouke Verbree, Erik Jan Marinissen, Philippe Roussel, and Dimitrios Velenis. 2010. On the Cost-Effectiveness of Matching Repositories of Pre-tested Wafers for Wafer-to-Wafer 3D Chip Stacking. In *IEEE European Test Symposium*.

[61] C. G. Willard, A. Snell, and M. Feldman. 2015. HPC Application Support for GPU Computing. http://www.intersect360.com/industry/reports.php?id=131. (2015). [Online; accessed 2017-04-04].

[62] Amir Kavyan Ziabari, José L Abellán, Yenai Ma, Ajay Joshi, and David Kaeli. 2015. Asymmetric NoC Architectures for GPU Systems. In *Proceedings of the 9th International Symposium on Networks-on-Chip*. ACM.

[63] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. 2016. UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* (2016).