Beyond the Socket: NUMA-Aware GPUs

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 leverage strong data parallelism exposed via the programming model attached to a uniform memory access system (UMA) With Moore's law slowing, for GPUs to continue scaling performance (which largely depends on SIMT core count) they are likely embrace multi-socket designs where more transistors are more readily available. However when moving to multi-socket designs, maintaining the illusion of a uniform memory system becomes increasingly difficult. In this work we investigate future multi-socket NUMA GPU designs and show that significant changes are needed to the GPU interconnect and caching systems 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 to minimize the impact of non-uniform memory access (NUMA) effects. Our NUMA-aware GPU is able to outperform a single GPU by XXX%, XXX%, and XXX% and achieves XXX%, XXX%, and XXX% of theoretical application scalability in 2, 4, and 8 socket designs respectively. Implementable today, NUMA-aware multi-socket GPUs may be a promising candidate for scaling GPU performance beyond a single die.

1. INTRODUCTION

In the last 10 years GPUs computing has transformed the high performance computing, machine learning, and data analytics fields that were previously dominated by CPU-based installations [1,2,3,4]. Modern systems now rely on a combination of GPUs and CPUs to leverage high throughput data parallel GPUs along side latency critical execution occurring on CPUs. In part, GPU-accelerated computing has been successful in these domains because of native support for data parallel programming languages [5, 6] that reduce programmer burden when trying to scale programs across ever growing data sets.

Nevertheless, with GPU dies nearing the reticle limitation of maximal die size and transistor density growth rate slowing down [7], programmers 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 leveraging special mechanisms such as Peer-2-Peer access [8] or a combination of MPI and CUDA [9] to manage multiple GPUs. While these programming extensions enable advanced programmers to utilize more than one GPU for high throughput computation, they require re-writing of traditional single-GPU application which may slow their adoption rate.

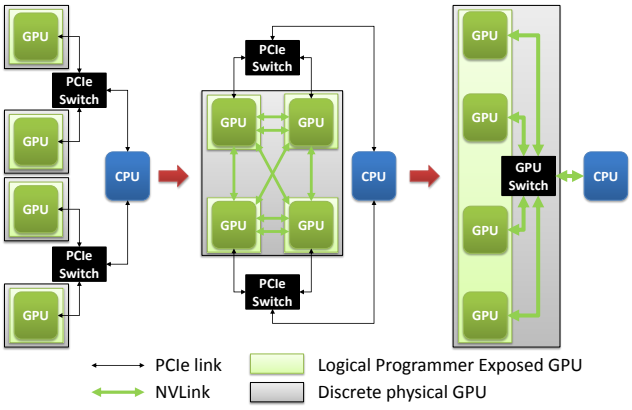

Figure 1: The evolution of GPUs from discrete pluggable PCIe devices to high pin-count multi-socket accelerators utilizing switched interconnects.

Recently GPUs have started transitioning from using the PCIe peripheral interface to having single protocol interconnections between both the GPUs and CPUs [10, 11, 12], as shown in Figure 1. As a result the physical interface of GPUs is transitioning away from a discrete pluggable PCIe card into a high pin-count socketed processor similar to CPU designs. The socketed interface is required not only for power delivery but because the GPU and CPU interconnects require printed circuit board (PCB) level integration to provide the needed interconnect bandwidth that can be an order of magnitude higher than PCIe bandwidths. The evolution of GPUs from externally connected devices to high connectivity multi-socket designs is a natural progression as interconnect bandwidth becomes a performance critical system component.

The onset of multi-socket GPUs provides a pivot point for GPU and system vendors. On one hand, they can continue to expose multi-socket GPUs as individual GPUs and force programmers to use multiple programming paradigms to leverage multiple GPUs. On the other, vendors may choose to expose multi-socket designs as a single non-uniform

memory access (NUMA) GPU resource. 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 single-GPU programming model that GPU developers have become accustomed.

Prior work has examined aggregating multiple GPUs together under a single programming model [13, 14], however much of this work was done in an era where GPUs had limited memory addressability and relied on high latency, low bandwidth PCIe interconnects. As a result, prior work focused primarily on improving the multi-GPU programming paradigm rather than achieving highly scalable performance. However today, in the the era of unified virtual addressing [15], cache line addressable high bandwidth interconnects [16], and dedicated CPU–GPU socketed PCB designs [11], scalable multi-GPU performance may be achievable.

Building upon prior work, we propose a multi-socket NUMA-aware GPU architecture and runtime that aggregates multiple-GPUs into a single large GPU and achieves good performance scalability across a wide range of benchmarks. First we apply several known NUMA techniques from the CPU community to enable a locality aware multi-GPU runtime but show these optimizations are not enough to achieve performance scalability in a multi-socket NUMA-GPU. We then propose interconnect and cache improvements for GPUs to make their architecture *NUMA-aware* and show that with these improvements multi-socket GPUs can achieve good performance scalability when executing programs optimized for a legacy single-die GPU. In this work, we make the following contributions:

- We show that traditional NUMA memory placement and scheduling policies are not sufficient for multi-socket GPUs to achieve good performance scalability and then demonstrate that intersocket bandwidth will be the primary performance limiter in NUMA-GPUs.
- By exploiting program phase behavior we show that intersocket links (and thus bandwidth) should be dynamically varied at runtime to maximize link utilization. Moreover, we show that dynamic policy must be determined on a per GPU basis, as a global policies will fail to capture per GPU phase behavior.
- We show that both L1 and L2 caches within the GPU must be made NUMA-aware and dynamically vary caching policy to minimize multi-socket NUMA effects. We show that the benefit of extending cache coherence across multiple GPU sockets far outweighs the overhead of a course grained coherence implementation.
- Using a NUMA-aware runtime and GPU microarchitecture we demonstrate that future multi-socket GPUs may allow traditional GPU programs to scale efficiently to as many as 8 GPU sockets, providing significant headroom before programmers must re-architect applications to obtain performance scalability.

2. MOTIVATION AND BACKGROUND

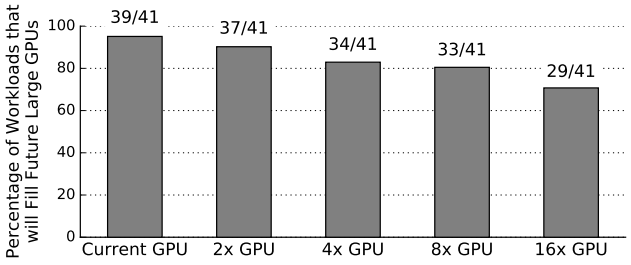

Figure 2: Percentage of benchmarks that have more CTAs on average than available SMs.

Over the last decade single GPU performance has scaled well due to a significant growth in per-GPU transistor count and DRAM bandwidth. For example, in 2010 Nvidia's Fermi GPUs integrated 1.95B transistors on a 529mm^2 die, with 180GB/s of DRAM bandwidth. In 2016 Nvidia's Pascal GPUs contained 12B transistors on a 610 mm^2 die, while relying on 720GB/s of main memory bandwidth. Unfortunately transistor density is slowing significantly and vendors are not providing roadmaps beyond 7nm. Moreover, GPU die sizes which have been also increasing generationally, are expected to slow down due to limitations in lithography and manufacturing cost.

Without either larger or denser dies, manufacturers must turn towards alternative solutions to significantly increase GPU performance. Recently 3D die-stacking has seen significant interest due to its successes in the high bandwidth DRAM design space [17]. Unfortunately 3D die-stacking still has a significant number of engineering challenges related to power delivery, energy density, and cooling [18] when employed in power hungry, maximal die-sized chips such as GPUs. Thus we believe 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 PCIe attached peripheral to the primary computing component at system design time. As a result, these GPU optimized systems employ custom PCB designs that accommodate high pin count socketed GPUs [10] with inter-GPU interconnects resembling QPI or Hypertransport [16, 19, 20]. However, despite the rapid improvement in hardware capabilities, these GPU optimized systems have thus far exposed the GPUs in the system as individually addressable GPUs. These multi-GPU systems can provide high aggregate throughput, for multiple concurrent GPU accelerated programs, but to accelerate a single GPU program they require layering additional software runtimes on top of native GPU programming interfaces such as CUDA or openCL [6, 21]. By requiring significant application redesign many applications are never ported to take advantage of multiple GPUs.

While extending single GPU application performance significantly is a laudable goal, we must first understand if applications will be able to leverage future larger GPUs. Today NVIDIA's largest GPUs contain 54 SMs with growth to 64

SMs likely as the product line matures. Across a benchmark set of 41 applications, later described in Section 3.2 and Table 2, we find that many single-GPU optimized workloads contain enough data parallelism to fill future GPUs that may be 2, 4, or 8× larger than today's biggest GPUs. While single-GPU optimized applications are unlikely to scale to tens of thousands of GPUs across an entire data center, for the majority of GPU programs written today we believe there will be significant demand for GPUs that can scale application performance by as much as a factor of 8× while requiring no additional software engineering.

To understand the performance scalability of a multi-socket NUMA GPU we examine the performance that a future 4-socket GPU can achieve while executing workloads optimized for a modern single GPU. Not surprisingly, when executing uniform memory access (UMA) optimized GPU programs on a highly NUMA GPU, performance does not scale well. We identify that despite software optimizations for locality, interconnect bandwidth in a NUMA GPU is still the primary limiter on scalable performance. To overcome this bottleneck we propose two classes of improvements that leverage application phasing to reduce the observed NUMA penalty. First, in Section 4 we examine the ability of switch connected GPUs to dynamically change their ingress and egress links from symmetric bandwidth provisioning to asymmetric bandwidth provisioning. By using existing interconnects more efficiently the effective NUMA bandwidth ratio of remote memory to local memory decreases, improving performance.

Second, we propose that to minimize traffic on oversubscribed communication links GPU caches need to become NUMA-aware. Traditional on-chip caches are optimized to maximize overall hitrate, thus minimizing off-chip bandwidth. When on-chip caches are backed by a single memory system, or even two symmetric memory systems, maximizing hitrate is the best objective function for cache policy. However, in highly NUMA systems, not all cache misses have the same relative cost. Specifically, due to the NUMA penalty of accessing remote memory, overall application throughput may be maximized by preferencing cache capacity (and thus improving hit rate) towards data that resides in remote NUMA zones rather than faster local memory. To this end, we propose new NUMA-aware GPU caches that dynamically skew cache capacity when it is detected that either the remote or local memory bandwidth is oversubscribed.

By architecting NUMA-aware GPUs we show that single-GPU application performance can be scaled up significantly before costly application re-writes may be required. For existing systems like Nvidia's 8-GPU DGX Supercomputer [10], applications which execute on, at most, one GPU could see up to an 8× performance increase transparently, requiring no application programmer effort. Before diving into microarchitectural details and results, we first describe the transparent multi-socket GPU runtime that enables such an 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, threadblock (CTA) scheduling policies need not be sophisticated to capture locality, which which has been destroyed by the memory system layout. For future NUMA GPUs to work well, both 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 it is possible to design a framework and a runtime system that transparently decomposes GPU kernels in sub-kernels and executes them on multiple PCIe attached GPUs in parallel. 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. Special care needs to ensure that per-GPU memory fences are promoted to system level and seen by all GPUs as well as guaranteeing that sub-kernels CTA identifiers are properly managed to reflect the barriered completion state of all sub-kernels before the original kernel call is completed. In [14] these two problems were solved by introducing code annotations and an additional source-to-source compiler which was also responsible for statically partitioning data placement and computation.

In our work, we follow a similar strategy but without using an source to source compiler. Unlike prior work, we are able to rely on NVIDIA's Unified Virtual Addressing [15] to allow dynamic placement of pages into memory at runtime rather than static memory placement. Similarly, technologies like cache line granularity interconnects like NVIDIA's NVLINK [16] 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 executions 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 undesireable 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 [22].

On a single-GPU fine grain 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

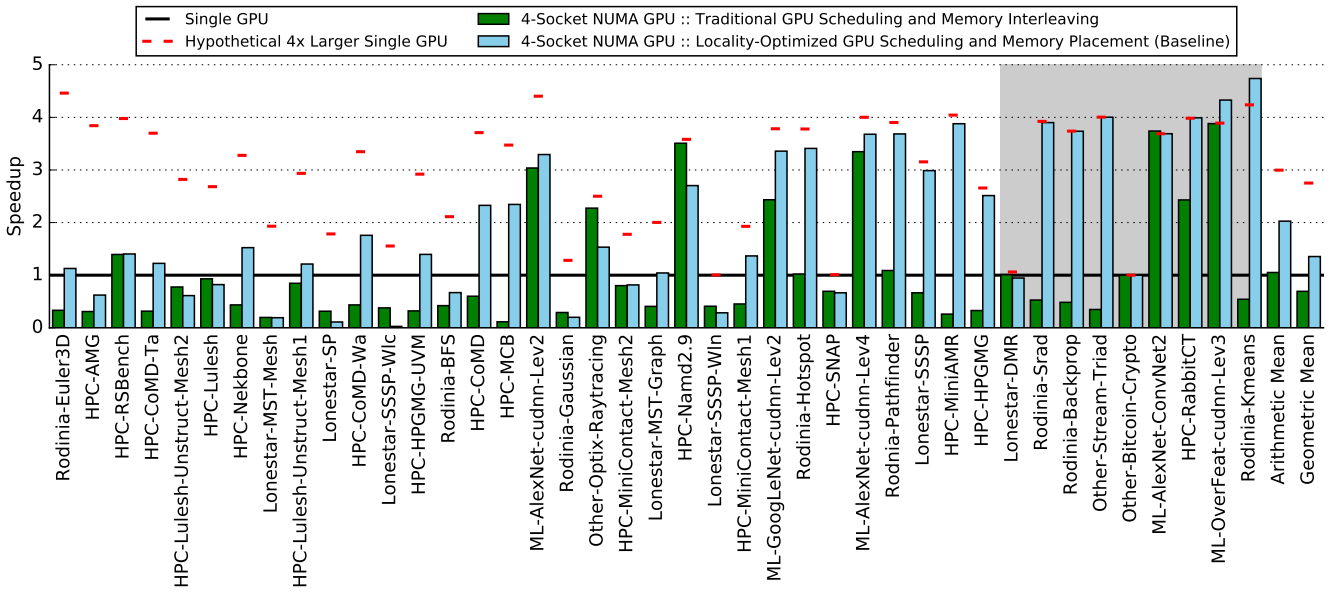

Figure 3: Relative performance of a 4-socket NUMA GPU using to a single GPU and a hypothetical 4× larger (all resources scaled) single GPU showing upper bound of performance this application can achieve via GPU hardware scaling. For the Locality-Optimized design, applications shown in grey achieve already 99% of theoretical scaling (*red dash*) without micro-architectural modification.

this penalty the GPU runtime much launch a block of CTAs to each GPU-socket at 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 has been shown to preserve data locality present in applications where contiguous CTAs also access contiguous memory regions [14, 22].

3.1 Performance Through Locality

Figure 3 we show the relative performance of a 4-socket NUMA GPU with respect to a single GPU under 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. We can clearly see that 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 even match the performance of a single GPU despite the large increase in hardware resources. Thus, using variants of prior proposals [14, 22], 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 are not scaling well on our proposed NUMA GPU system. To illustrate this, Figure 3 shows the workload throughput achievable by a hypothetical (unbuildable) 4× larger GPU with a red dash. This red dash represent an approximation of the maximum theoretical performance we could expect 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 4x larger GPU due higher cache hitrates because contiguous block scheduling is more cache friendly than traditional GPU scheduling.

However, for the applications on the left side there is 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 amount of remote data accesses still occurring. Because our goal is to provide scalable performance for single-GPU optimized applications, in the rest of the paper we aim to close this performance gap through microarchitectural innovation. To simplify later discussion, we choose to exclude benchmarks that achieve $\geq$99% of the theoretical performance with SW-only locality optimizations, however we include all benchmarks in our final results to show the overall performance scalability achievable with NUMA-aware multi-socket GPUs.

3.2 Simulation Methodology

To evaluate the performance of multi-socket GPUs we use a proprietary, cycle-level, trace-driven simulator for GPU

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

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

systems running single GPU CUDA applications. For our baseline, we model a single GPU that approximates the latest NVIDIA Pascal architecture [23]. Each of the Streaming Multiprocessors (SM) is modeled as an in-order processor with multiple levels of cache hierarchy containing private, per-SM, L1 caches and multi-banked, shared, L2 cache. Each GPU is backed by its local high bandwidth DRAM memory. Our multi-socket GPU system contains four of these GPUs interconnected through a full bandwidth GPU switch as shown on Figure 1. Table 1 stands as an overview of the simulation parameters.

We study our proposal using 41 workloads taken from a broad range of production codes based on the HPC CORAL benchmarks [24], graph applications from Lonestar [25], compute applications from Rodinia [26], in addition to 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, scientific computing, etc. We run our simulations until the completion of the entire applications or the number of kernels shown on Table 2. Table 2 shown after the conclusions is provided to show more complete data about each workload including average number of CTAs for each application (weighed by the time spent on each kernel) and the memory footprint in MB.

4. ASYMMETRY-AWARE INTERCONNECTS

Figure 4(a) shows a generic multi-socket system with symmetric bandwidth assignments at each inter-socket communication link. Static link capacity assignment at design time is very common and has multiple advantages. For example, in such case only one type of I/O circuitry (egress drivers or ingress receivers) along with only one type of control logic need to be implemented at each on-chip interface. Moreover, multi-socket switches result in simpler designs that support a statically provisioned worst-case bandwidth scenario. On the other hand, multi-socket link bandwidth has a very important impact on overall system performance.

Benchmark	Kernels	Time-weighted Average CTAs	Memory (MB)
ML-GoogLeNet-cudnn-Lev2	1	6272	1205
ML-AlexNet-cudnn-Lev2	1	1250	832
ML-OverFeat-cudann-Lev3	1	1800	388
ML-AlexNet-cudnn-Lev4	1	1014	32
ML-AlexNet-ConvNet2	1	6075	97
Rodinia-Backprop	2	4096	160
Rodinia-Euler3D	346	1008	25
Rodinia-BFS	24	1954	38
Rodinia-Gaussian	510	2599	78
Rodinia-Hotspot	1	7396	64
Rodinia-Kmeans	3	3249	221
Rodnia-Pathfinder	20	4630	1570
Rodinia-Srad	4	16384	98
HPC-SNAP	118	200	744
HPC-Nekbone-Large	300	5583	294
HPC-MiniAMR	33	76033	2752
HPC-MiniContact-Mesh1	500	250	21
HPC-MiniContact-Mesh2	127	15423	257
HPC-Lulesh-Unstruct-Mesh1	2000	435	19
HPC-Lulesh-Unstruct-Mesh2	200	4940	208
HPC-AMG	88	241549	3744
HPC-RSBench	1	7813	19
HPC-MCB	1	5001	162
HPC-NAMD2.9	1	3888	88
HPC-RabbitCT	1	131072	524
HPC-Lulesh	105	12202	578
HPC-CoMD	350	3588	319
HPC-CoMD-Wa	350	13691	393
HPC-CoMD-Ta	350	5724	394
HPC-HPGMG-UVM	359	10436	1975
HPC-HPGMG	317	10506	1571
Lonestar-SP	11	75	8
Lonestar-MST-Graph	87	770	86
Lonestar-MST-Mesh	71	895	75
Lonestar-SSSP-Wln	1000	60	21
Lonestar-DMR	3	82	248
Lonestar-SSSP-Wlc	1300	163	21
Lonestar-SSSP	102	1046	38
Other-Stream-Triad	5	699051	3146
Other-Optix-Raytracing	1	3072	87
Other-Bitcoin-Crypto	1	60	5898

Table 2: Application footprint and average number of CTAs (thread blocks) available during time-weighted execution.

For the applications which saturate links between sockets, doubling NVLink capacity potentially achieves a 2× speedup. As I/O bandwidth is a very limited and expensive resource, this result motivates us to look for alternatives that keep wire and I/O resources at very high utilization.

We note that the typically employed symmetric link capacity assignments may result in lower overall utilization of the link wires in many cases. Moreover, in case of multi-socket NUMA GPU systems, we do observe that there are applications in which some GPUs have a very different utilization of its egress and ingress channels in different phases of execution. Figure 5 shows dynamic link utilization for a snapshot of `HPC-HPGMG-UVM` application running on our baseline 4 GPUs TMS-GPU system. Vertical dotted black lines represent the beginning kernel calls that are split across 4 sockets as explained in Section 2. We can see that few initial small kernels have a negligible interconnect utilization on all links. However, for the the upcoming larger kernels at the figure, GPU0 and GPU2 fully saturate their ingress links,

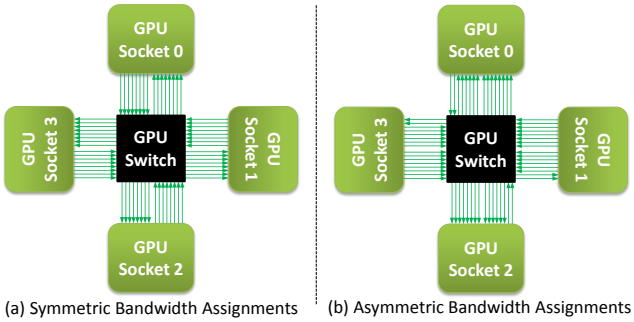

Figure 4: Multi-socket GPU systems with symmetric and assymeteric capacity assignments.

while GPU1 and GPU3 fully saturate their egress links. At the same time GPU0 and GPU2 pair, and GPU1 and GPU3 pair barely use their egress or ingress links accordingly.

In the rest of the workloads we observe other inter-socket traffic patterns that would prefer a disproportional bandwidth distribution. A common scenario is writing to the same memory section by all CTAs at the end of a kernel (parallel reductions, data gathering). For CTAs running on one of the sockets, GPU0 for example, those memory instructions are local and does not produce any traffic through the inter-socket interconnection network. Contrarly, CTAs dispatched to other GPUs issue remote memory writes, saturating their egress links while ingress links remain underutilized. Such a communication pattern utilizes only 50% of available bandwidth. Increasing the number of ingress links for GPU0 (by turning around direction of egress ones), and switching the direction of ingress channels for all the other GPUs, we potentially imporve the performance by enhancing the usage of NVLink.

Motivated by these findings, we propose to dynamically control multi-socket link bandwidth assignments in each direction on a per-GPU basis resulting in dynamically asymmetric link capacity assignments, as shown in Figure 4(b). Such dynamic and asymmetric link capacity distribution is expected to yield higher wire utilization and essentially higher effective bandwidth and performance for a given set of I/O resources. This mechanism is somewhat similar to DRAM interface for example, where the same set of wires is used for both read and write directions interchangeably, and link direction is reversed based on a dynamic state of the system [].

For basic link configuration we assume and model point-to-point links with multiple lanes each, similarly to NVLink sub-links [23]. In such link, 8 lanes with 8GB/s capacity per lane results in an aggregate bandwidth of 64GB/s in each direction. We propose an adaptive asymmetric scheme that works as following. For each link in the system, at kernel launch we start with symmetric link assignments with 8 lanes per direction. Then, we periodically sample 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, we reverse the direction of one of the unsaturated lanes. Repeating the previous step and sampling the lanes status, we stop either when equilibrium is reached, or alternatively all the lanes but one have been reversed (we always keep a minimum bandwidth of 8GB/s in each direction). There are two important factors that characterize the behaviour above (i) `SampleTime`: the frequency at which the scheme samples for a possible reconfiguration and (ii) `SwitchTime`: the cost of switching the direction of a lane. Switching period is comprised of the time it takes to drain all the pending packets in a given direction, and the time it takes to physically reconfigure the lane to start transmitting in the opposite direction. —— Evgeny : still need to add what are teh disadvnatages of assymetric (double IO interfaces and switching complexity)

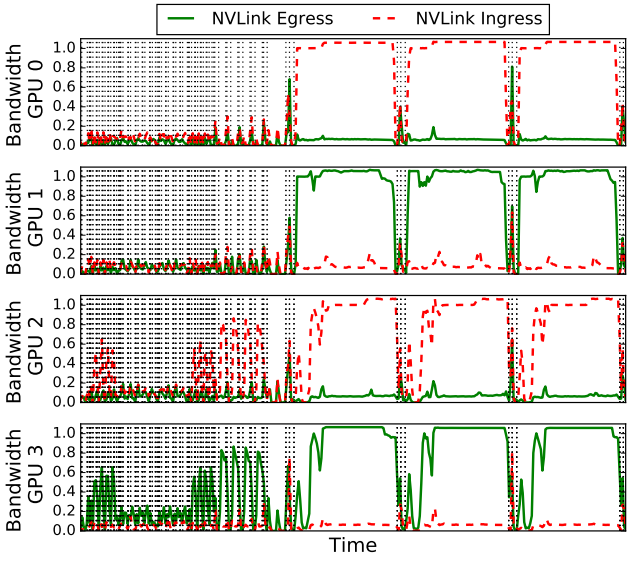

Figure 5: Normalized NVLink bandwidth profile for `HPC-HPGMG-UVM` showing example of asymmetric link utilization between GPUs and within a GPU depending on kernel and application phasing.

4.1 Results

Figure 6 shows the performance improvement, with respect to our baseline architecture by exploring different values of the `SampleTime` and assuming a `SwitchTime` of 100 cycles as reported in [**?**]. Also, Figure 6 gives an upper-bound performance when doubling the available interconnect bandwidth to 256GB/s (128GB/s per direction). For the benchmarks on the right side, where the baseline TMS-GPU performs close to 4× larger single GPU, increasing the inter-socket bandwidth does not improve performance significantly. Contiguous CTA scheduling and first-touch memory placement preserves good data locality in those cases, so that both ingress and egress links are underutilized. On the left side, we can see that for some applications, 2× NVLink bandwidth translates to 2× speedup, while dynamic lane switching achieves up to 80% performance improvement (`Lonestar-SP`). On average, double the interconnect bandwidth gives 50% while our dynamic solution achieves 15% improvements. We can also see that the sampling frequency plays a critical role and a too large sample time doesn't capture application dynamics resulting in smaller improvements. We find 5K cycles to be a reasonable sam-

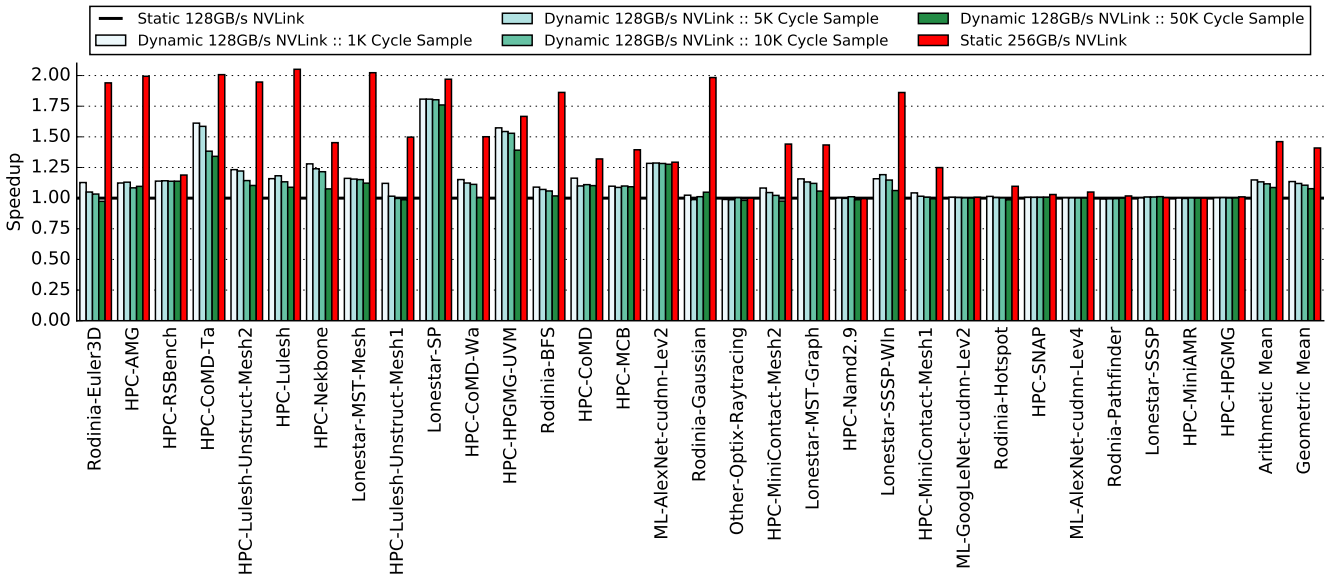

Figure 6: Relative speedup of the dynamic NVlink adaptivity with respect to the baseline architecture by varying sample time and assuming switch time of 100 cycles. In red, relative speedup achievable by doubling the bandwidth.

pling period able to achieve good results without degrading performance.

For benchmarks like `Rodinia-Euler-3D`, `HPC-AMG`, and `HPC-Lulesh`, doubling the NVLink bandwidth provides $2\times$ speedup, while our proposed dynamic link assignment mechanism is not able to significantly improve the performance. Those are the workloads that saturate both link directions, so there is no opportunity to provide additional bandwidth by turning links around. For the rest of the applications on the left side, performance improvement is directly proportional to the actual link utilization of unsaturated lanes. For example, consider a case where a GPU fully saturates its egress links and incoming traffic achieves only 75% of ingress bandwidth (using 6 out of 8 lanes). Then, adaptive policy is able to increase the outgoing bandwidth for a maximal 25% by turning around those 2 unused ingress lanes. If such a traffic communication pattern is sustained throughout 50% of the total execution time, the maximal performance improvement that our dynamic link assignment can achieve is 12.5%.

Another important parameter is switching period needed to turn around a single lane direction. Our results point that higher `SwitchTime` values, like 200 and 500 cycles, degrade the average performance by less than 2% compared to 100 cycle switching period. We observe the maximum slowdown of 25% in case of `HPC-CoMD-Ta` and `HPC-CoMD-Wa` when switch time reaches 500 cycles. At the same time, faster lane switch (10 cycles) does not improve the performance compared to 100 cycles. Thus from now, we define our adaptive link assignment policy which samples saturation status on every 5K cycles and eventually turn link direction with 100 cycles penalty.

5. NUMA-AWARE CACHE MANAGEMENT

In Section 4 we have shown that intersocket bandwidth is one of the most important factors in achieving scalable NUMA-GPU performance. By dynamically retraining the link to support asymmetric bandwidth configurations, rather than a stati uniform balance, we are able to improve the performance of applications by 15%. This improvement comes not from increasing the absolute bandwidth of the link, but by using the existing link more efficiently. 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, this technique yields little improvement. To improve performance in situations where dynamic link balancing is not effective, system designers can increase link bandwidth, which is very expensive, or try and decrease the amount of traffic that crosses the low bandwidth communication channels. To decrease off-chip memory traffic, architects typically turn to caches to capture locality whenever possible.

GPU cache hierarchies differ from traditional CPU hierarchies where in they are not supported by strong hardware coherence protocols [**?**]. They also differ from CPU protocols in that caches designed for a single uniform memory access GPU 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 shown in Figure 7(a), a GPU today is typically composed of relatively large SW managed coherent L1 caches that are physically proximal to the SM's, while a relatively small, distributed, non-coherent L2 cache resides memory side 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. By then placing the L2 caches memory-side, they do not need to participate in the coherence protocol. Because of the fine grained interleaving

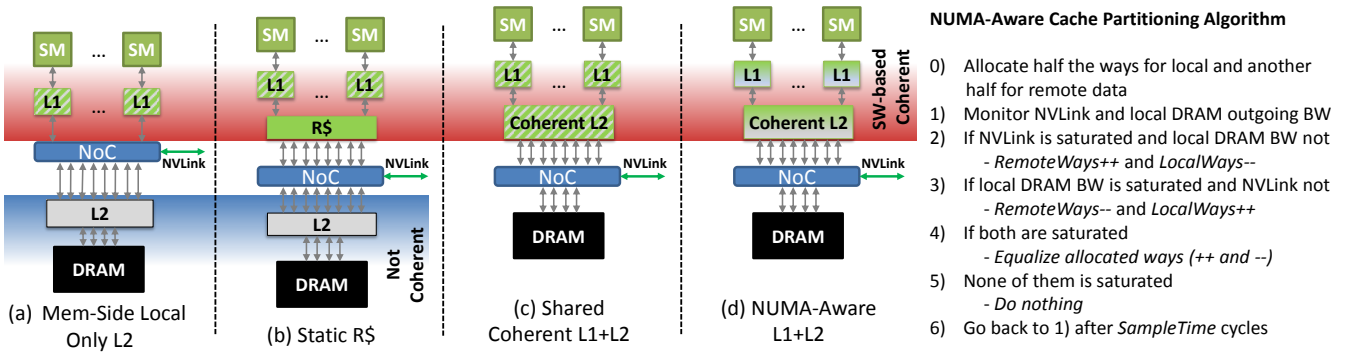

Figure 7: Potential L2 cache organizations to balance capacity between remote and local NUMA memory systems.

of physical memory addresses across memory channels, the L2 caches are inherently load balanced.

5.1 Cache Partitioning Options

In NUMA-designs interleaving memory references across the low bandwidth NUMA interconnections results in poor performance, as shown in Figure 3. Similarly, in NUMA-GPUs utilizing the traditional memory side L2 caches that depend on fine grained memory interleaving is a bad decision because these memory side caches are not able to cache remote memory, and thus cannot 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 [22] for caching remote data only (we denote them as Remote caches here). By balancing this capacity between local and remote caches they limit the need for invalidating the entire L2 cache capacity when extending the GPU coherence into the R$ cache while still minimizing crossbar bandwidth. An example of this organization is shown in Figure 7(b).

Static designs that allocate cache capacity to local memory, remote memory, or any other balance, may achieve reasonable performance but they lack flexibility. Much like application phasing was shown to affect NUMA bandwidth needs, 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 sockets within the NUMA-GPU to access data locally, while others are accessing remote data, as shown in Figure 5, allowing the entire L2 to cache both local and remote data should outperform any static policy. Second, while we show that many applications will be able to completely utilize large NUMA-GPUs, this may not always be the case. GPUs within the data center are being virtualized and there is on-going work to support concurrent execution of multiple kernels and/or processes within a single GPU [**?**]. If a large NUMA-GPU is subpartitioned, it is intuitive that system software attempt to partition it along the NUMA boundaries so that a single small application need not endure NUMA effects. To effectively capture locality in such a situation, NUMA-aware GPUs need to be able to flexibly move from 100% local to 100% remote caching at runtime, rather than be statically partitioned at design time.

Static L2 cache partitioning between local and remote data results in a GPU cache organization in which the L1 caches contain both local and remote data but the GPU L2 cache(s) will contain only local or remote data. To-date, single socket GPUs have not moved their memory-side caches to processor side because the overhead of cache invalidation (due to coherence) is an unnecessary performance penalty, with no performance upside to justify the cost. However in a multi-socket NUMA GPU, rather than statically partitioning the L2 cache at design time, the entire L2 cache could be made an extension of the L1 and both local and remote data could dynamically contend for capacity. Figure 7(c) shows a configuration with such coherent L2 cache. While conceptually simple, allowing both remote and local memory accesses to contend for cache capacity in a NUMA system is flawed.

In non-NUMA systems, performance is maximized by optimizing for cache hitrate, which minimizes off-chip memory system bandwidth. In NUMA systems however, not all cache misses have the same relative cost, and thus performance impact. A cache miss to a NUMA-local memory address will have a smaller cost (in both terms of latency and bandwidth) than a cache miss to a NUMA-remote memory address. Thus, it should be beneficial to dynamically skew cache allocation policy to preference caching of remote memory over local data when it is determined the system is bottle necked on NUMA bandwidth.

We propose that rather than allowing both the L1 and L2 caches to be contended by local and remote memory accesses, both levels of the GPU cache hierarchy become NUMA-aware and dynamically adjust their cache resources to preference caching of remote NUMA data when it is determined that the NVLINK bandwidth in our hypothetical GPU is oversubscribed. We do this by implementing the NUMA-aware cache partitioning algorithm, similar to the adaptive NVLink switching policy. Cache organization and a breif summary is shown on Figure 7(d).

At the beginning, we allocate one half of the cache ways for local, and another half for caching remote data (Step ⓪). After the sampling period, we collect the bandwidth values on the outgoing NVLink and local memory channels. Link utilization above 99% is considered to be saturated (Step ①). In case where NVLink is saturated and local memory

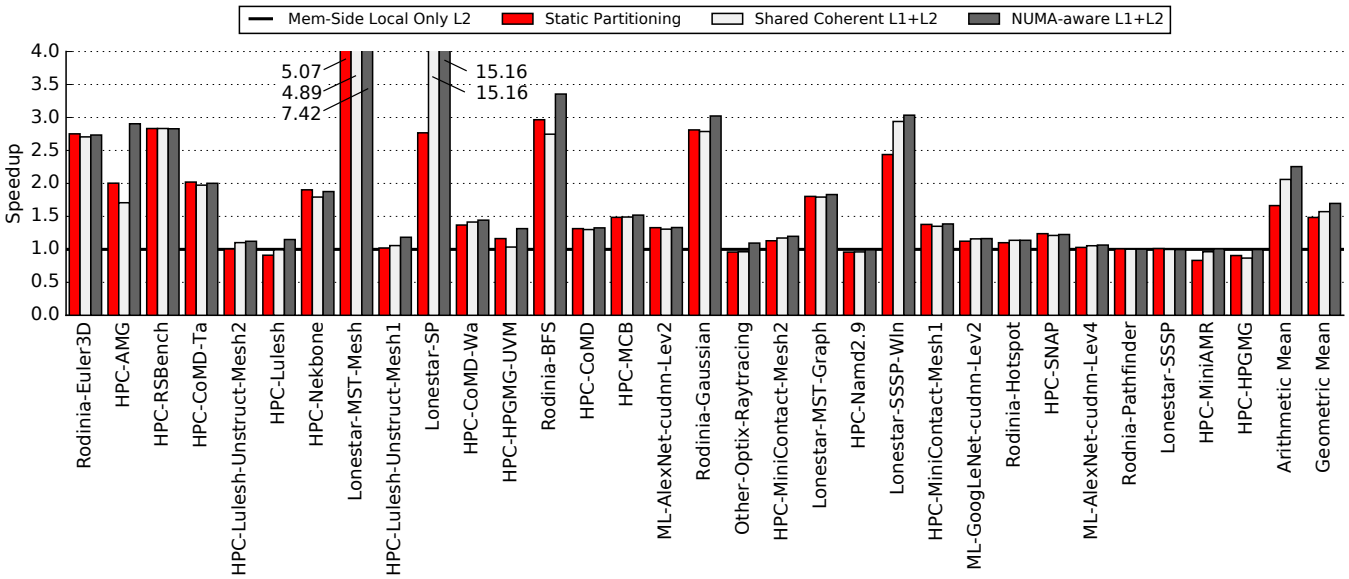

Figure 8: Performance of NUMA-aware dynamic cache partitioning in a 4-socket GPU compared to memory-side L2 and previously proposed static partitioning.

bandwidth is not, the partitioning algorithm tries to reduce remote traffic by assigning one way from a set of local ways, and allocating it to a set of remote ways (Step ②). If it is a write-back cache (only L2), dirty lines from the assigned way are written back to the backing memory. Similarly, if the local memory BW is saturated and NVLink is not, the policy takes one way from remote, and allocates it now to a set of local ways (Step ③). That way, our NUMA-aware cache partitioning algorithm reduces remote or local link utilization depending on which one of the two stands as a performance bottleneck. In case where both NVLink and local memory channels are saturated, we gradually equalize the number of ways assigned for remote and local cache lines (Step ④). Finally, if none of the links have been saturated, the policy takes no action (Step ⑤).

5.2 Results

Figure 8 compares the performance of different cache partitioning configurations. Our baseline is a 4-socket GPU with memory side, local-only L2 caches. With *red bars*, we show speedup gained by equally splitting the L2 cache budget, between the gpu-side R$ which caches only remote data, and mem-side L2. Previous work reports this as the best static L2 partitioning scheme [22]. Indeed, this configuration improves the performance by 50% on average, although for some benchmarks, like `HPC-MiniAMR` and `HPC-HPGMG` it hurts the performance by up to 10%. Those workloads stress local memory bandwidth and have negligible inter-socket traffic, thus lowering the mem-side L2 capacity creates the bottleneck. *Light gray* bars stand for configuration with shared and coherent L2 caches where both local and remote data contend for the available capacity. On average, it outperforms static cache partitioning, although looking at the particular benchmarks, the conclusion is not clear.

Finally, *gray* and *dark gray* bars stand for our dynamic NUMA-aware cache partitioning policy which additionaly improves the average performance. All caches in the NUMA-aware multi-socket GPU system are able to independently change their local behavior based on the hardware counters monitoring the outgoing NVLink and local memory bandwidth. The flexibility of our proposal and its ability to adapt at runtime, results in the best posible configuration a particular benchmark can exploit. For the workloads on the left side of Figure 8 which fully saturate the NVLink bandwidth, NUMA-aware dynamic policy arrange the L2 caches to be entirely used as a remote-only cache. Unlike the conclusions of the previous work, allocating 100% of the cache to remote data is by far the highest performing configuration for these applications. The difference in this conclusion is likely due to the fact that their intra-GPU crossbar has significantly more bandwidth than our multi-socket NUMA links, and thus we need to dedicate more (all) of the GPU's L1 and L2 capacity to caching remote data, despite the fact that SW based cache coherence will now effectively flush the entire L2 cache on all GPUs when those operations occur. Moreover, workloads on the right side expose good locality, thus prefer L1 and L2 caches to store (mostly) local data, something that NUMA-aware policy can adapt to. We see that dynamic partitioning can improve average GPU performance by 71% compared to the memory side L2, and 21% compared to the static cache partitioning.

Extending NUMA-aware dynamic cache mechanism to L1 caches results in an interesting observation. In situations where both L1 and L2 caches end up as remote-only (finding that NVLink is saturated while local memory bandwidth is not), we have detected performance degradation for some benchmarks. With no local ways left inside the L1 caches, local memory accesses start missing in the L1, reserving all MSHR entries. Running out of available MSHR entries blocks the entire cache thus none of the memory accesses can be served, reducing the number of available warps to keep an SM busy. On the other side, with L2 caches being

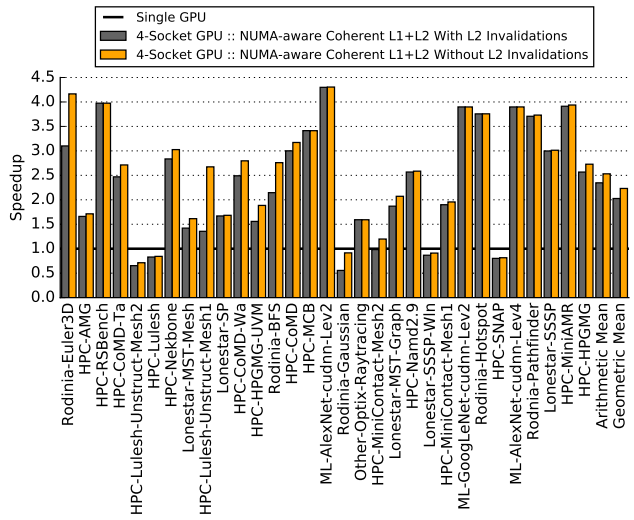

Figure 9: Performance overhead of extending current GPU software based coherence into the GPU L2 caches.

also remote-only, thess local memory misses are now propagated further to the local memory. That way, the execution time increases without enough parallelsim to hide prolonged memory accesses. To mitigate this problem, we tunned our NUMA-aware policy for L1 caches, by not allowing them to allocate all available ways to either local or remote data, but keep at least one way for any of both.

5.3 Extending the Coherency to L2 Caches

When extending the software controlled GPU coherence protocol into the GPU L2 caches, L1 coherence operations (flushes) must also be extended into the GPU L2 caches. To understand the impact of these coherence operations on our dynamic cache performance we evaluated a hypothetical cache implementation that need not flush on software defined boundaries. This experiment approximates a speed of light coherence implementation in which caches are still coherent but there is no cost for maintaining that coherence. In a 4-socket, 256 SM NUMA-GPU we observe that the coherence overhead of current software based GPU coherence is only XXX%. We conclude that despite some coherence overheads, the benefit of dynamic NUMA-aware coherent L2 caches on multi-socket GPUs will be required to maximize both performance and GPU flexibility.

6. DISCUSSION

In Sections 4 and 5 we explored the performance implications of dynamically managing GPU interconnect bandwidth and allowing individual GPUs to dymnamically allocate on-chip cache capacity to decrease dependance on memory system resources when oversubscribed. Both of these techniques aim to more efficiently utilize the available system bandwidth within our NUMA multi-GPU system, closing the gap between multi-socket bandwidth and local memory bandwidth. These two techniques are orthoginal and could be applied in isolation or in combination. Dynamic interconnect balancing has an implementation advantage in that the system level changes to enable this feature are very isolated from the larger GPU, only the link level load balancer on both the GPU and switch need to be modified. Conversely, enabling GPU caching of remote memory and dynamic balancing of cache capacity based on interconnect utilization requires changes to both the physical cache architectures and the GPU coherence protocol.

Because these two features target similar improvements, when employed together their effects are not strictly additive. Figure 10 shows the improvement when applying both dynamic interconnect and dynamic cache management together. For benchmarks such as `CoMD`, these features contribute nearly equally to the overall improvement, but for others such as `HPGMG` or `MST`, interconnect improvements or caching are the primary contributor respectively. On average, we observe that when combined we see XXX% improvement in multi-GPU performance when applying our proposed optimizations.

Scalability: When considering moving from a single socket GPU to multi-socket GPU solutions the question arises of what level of efficiency can be maintained with this approach. Figure 11 shows the scalability of a multi-socket GPU approach as we move from a single to eight socket multi-GPU implementation. While larger multi-socket GPUs may be possible to power and cool within a single node, we observe that due to parallel efficiency XXX, XXX, XXX.

Another paragraph here once we have results.

Multi-Tenancy of 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 4× larger than today's GPUs. With deep data becoming commonplace across many computing paradigms, we believe that the trend of having enough computation to saturate much larger single GPUs will continue into the forseeable future. However, when GPUs become larger at the expense of having multiple discrete GPUs within the system, questions related to GPU provisioning arise. Applications that cannot saturate such a GPU will leave resources underutilized and applications that may be running concurrently in the system currently have to coarse grain multi-plex the GPU in time cooperatively.

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 [**?**], spatial multi-plexing of a GPU [**?**], or through improved time division multiplexing with GPU pre-emptability [**?**]. To support improved per GPU efficiency, any of these solutions could be applied to a multi-socket GPU to improve utilization in cases where applications can not fill a significantly larger GPU. Alternatively, with additional software work multi-socket GPU designs should be able to be dynamically partitioned with granularity from 1–N logical GPUs if they are switch connected, providing yet another level of partition-

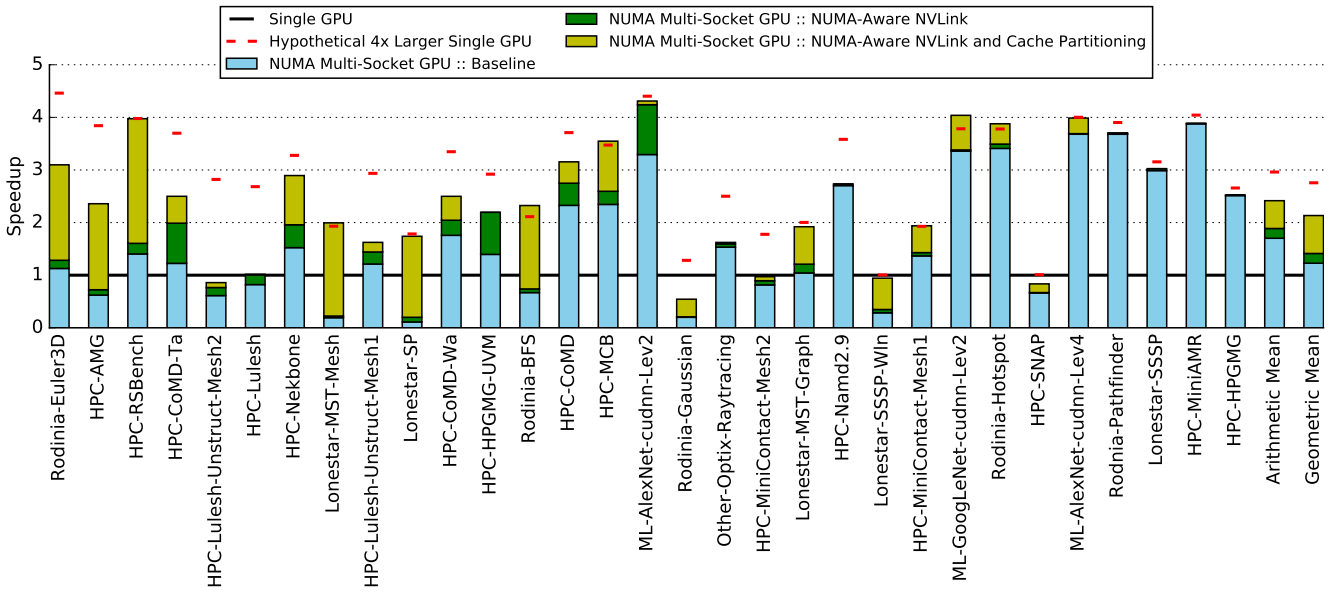

Figure 10: Performance of a 4-socket NUMA-aware GPU compared to a single GPU and a hypothetical 4x large single GPU with proportionally scaled resources.

able flexibility.

Power Implications: *Oreste or Evgeny* What happens to all the extra bandwidth we need over the interconnect and power. we should calculate the extra power but note that even using P2P GPU access there will still be many remote accesses. unfortunately we can't quantify that differential because (drives the point home) there are not multi-GPU versions for most of our 40+ benchmarks.

7. RELATED WORK

Multi-GPUs are widely used for scaling single GPU perfromance via integration of multiple GPUs at the system level [1, 10, 23, 27] for a rapidly growing pool of applications [2, 3, 4, 24]. Similarly, multi-socket and multi-node CPU installations have been employed and studied in context of HPC and data-center applications [28, 29, 30, 31]

Multi-GPU programming models require explicit programming of multiple GPUs using SW APIs such as Peer-2-Peer access [8] or a combination of MPI and CUDA [9] to manage multiple GPUs. These extensions require unique programming experience and non-negligible SW effort while adapting a single GPU application to take advantage of a Multi-GPU system. In this paper we execute single GPU application on a Multi-GPU system as if it was a single larger GPU via hardware innovations and extensions to the driver software to provide programmer- and OS-transparent execution, similarly to approaches proposed in the past [13, 14, 32].

Modern multi-socket CPU and GPU systems leverage most advanced interconnect technologies such as Nvidia Nvlink, Intel QPI and AMD Infinity [10, 12, 19]. These modern fabrics utilize high speed serial signalling technologies over unidirectional lanes collectively comprizing full-duplex links. This way each link capacity is statically allocated at design time and usually is symmetric in nature. In this paper we propose to dynamically re-allocate available link bandwidth resources by using same system wire resources and on-chip I/O interfaces, while implementing both recever and transmitter driver circuitry at each lane. This approach resembles previously proposed tri-state bi-directional bus technologies [33], or former technologies such as Intel front-side bus [34]. Our proposal however allows leveraging 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 context of CPU caches and QoS [35, 36, 37, 38, 39] For example, Rafique et al [37] proposed architectural support for shared cache management with quota-based approach. Qureshi et al [38] proposed to partition the cache space between applications. Jaleel et al [39] improved on this by proposing adaptive insertion policies. Recently, cache monitoring and allocation technologies were added to Intel Xeon procesors, targeted for QoS enforcement via dynamic repartitioning of on-chip CPU cache resources [36]. Efficient cache partitioning in GPU was primarly explored in context of L1 caches [40]. While dynamic cache paritioning was widely used for QoS and L1 utlization, to the best of our knowledge it was never used for . . .

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

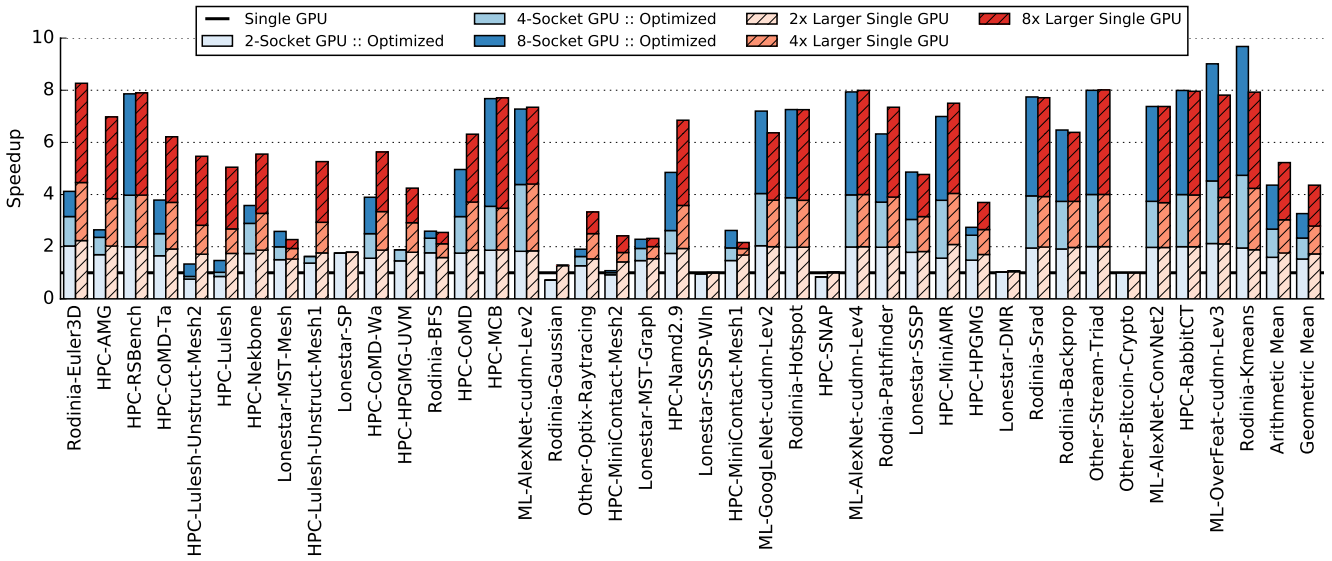

Figure 11: Performance scalability of a NUMA-aware GPU compared to theoretical maximim applications performance moving from 1 to 8 GPU sockets.

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 NUMA-aware multi-socket GPU is able to outperform current single GPUs designs by XXX%, XXX%, and XXX% and achieves XXX%, XXX%, and XXX% of theoretical application scalability in 2, 4, and 8 socket designs respectively. This scalable performance indicates that the challenges of designing a multi-socket NUMA GPU can be overcome through a combination of runtime and architectural optimization, making NUMA-aware GPUs a promising technology for scaling GPU performance beyond a single socket.

9. REFERENCES

[1] C. G. Willard, A. Snell, and M. Feldman, "HPC Application Support for GPU Computing," http://www.intersect360.com/industry/reports.php?id=131, 2015.

[2] NVIDIA, "NVIDIA cuDNN, GPU Accelerated Deep Learning," https://developer.nvidia.com/cudnn, accessed: 2016-11-17.

[3] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

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

[5] NVIDIA Corporation, "CUDA C Programming Guild v7.0," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015, [Online; accessed 09-May-2015].

[6] KHRONOS GROUP, "OpenCL 2.2 API Specification (Provisional)," https://www.khronos.org/opencl/, 2016, [Online; accessed 28-March-2017].

[7] P. Bright, "Moore's Law Really is Dead This Time," http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time, 2016, accessed: 2016-06-20.

[8] NVIDIA, "Multi-GPU Programming," http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf, 2011, [Online; accessed 28-March-2017].

[9] ——, "MPI Solutions for GPUs," https://developer.nvidia.com/mpi-solutions-gpus, 2016, [Online; accessed 28-March-2017].

[10] NVIDIA, "The World's First AI Supercomputer in a Box," http://www.nvidia.com/object/deep-learning-system.html, accessed: 2016-11-17.

[11] Lawerence Livermore National Laboratory, "CORAL/Sierra," https://asc.llnl.gov/coral-info, 2016, [Online; accessed 2-April-2017].

[12] AMD, "AMD's Infinity Fabric Detailed," http://wccftech.com/amds-infinity-fabric-detailed/, 2017, [Online; accessed 28-March-2017].

[13] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013.

[14] 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 *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, 2015.

[15] NVIDIA Corporation, "Unified Memory in CUDA 6," http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2013, [Online; accessed 28-May-2014].

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

[17] JEDEC, "High Bandwidth Memory(HBM) DRAM - JESD235," http://www.jedec.org/standards-documents/docs/jesd235, 2013, [Online; accessed 28-May-2014].

[18] J. Verbree, E. J. Marinissen, P. Roussel, and D. Velenis, "On the Cost-Effectiveness of Matching Repositories of Pre-tested Wafers for Wafer-to-Wafer 3D Chip Stacking," in *IEEE European Test Symposium*, 2010.

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

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

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

[22] 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 *International Symposium on Computer Architecture (ISCA)*, 2017.

[23] NVIDIA, "NVIDIA Tesla P100," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016, accessed: 2017-03-20.

[24] "Coral benchmarks," https://asc.llnl.gov/CORAL-benchmarks/, 2014.

[25] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular gpu kernels," in *International Symposium on Workload Characterization (IISWC)*, 2014.

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

[27] "Titan : The world's #1 open science super computer," https://www.olcf.ornl.gov/titan/.

[28] "The Xeon X5365," http://ark.intel.com/products/30702/Intel-Xeon-Processor-X5365-8M-Cache-3_00-GHz-1333-MHz-FSB, accessed: 2016-08-19.

[29] "IBM Power Systems Deep Dive," http://www-05.ibm.com/cz/events/febannouncement2012/pdf/power_architecture.pdf, 2012, accessed: 2016-08-19.

[30] "IBM zEnterprise 196 Technical Guide," http://www.redbooks.ibm.com/redbooks/pdfs/sg247833.pdf, 2011, accessed: 2016-08-19.

[31] "AMD Server Solutions Playbook," http://www.amd.com/Documents/AMD_Opteron_ServerPlaybook.pdf, 2012, accessed: 2016-08-19.

[32] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "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, 2015.

[33] M. M. Y. John R. Spence, "Clocked tri-state driver circuit," Patent US4 504 745, 1985. [Online]. Available: https://www.google.com/patents/US4504745

[34] Intel, "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, accessed: 2017-3-29.

[35] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," *Proceedings of ISC-2007*, June 2007.

[36] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel xeon processor e5-2600 v3 product family," *HPCA*, 2016.

[37] N. Rafique, W. Lim, and M. Thottethodi, "Architectural support for os-driven cmp cache management," *Proceedings of PACT-2006*, Sep 2006.

[38] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," *Proceedings of MICRO-2006*, Dec 2006.

[39] M. Q. J. S. S. S. J. A. Jaleel, W. Hasenplaugh and J. Emer, "Adaptive insertion policies for managing shared caches," *Proceedings of MICRO-2006*, Oct 2008.

[40] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Keckler, "Priority-based cache allocation in throughput processors," *HPCA*, 2015.