GPS: A Global Publish-Subscribe Model for Multi-GPU Memory Management

Harini Muthukrishnan
harinim@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Daniel Lustig
dlustig@nvidia.com
NVIDIA
Westford, Massachusetts, USA

David Nellans
dnellans@nvidia.com
NVIDIA
Austin, Texas, USA

Thomas Wenisch
twenisch@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

ABSTRACT

Suboptimal management of memory and bandwidth is one of the primary causes of low performance on systems comprising multiple GPUs. Existing memory management solutions like Unified Memory (UM) offer simplified programming but come at the cost of performance: applications can even exhibit slowdown with increasing GPU count due to their inability to leverage system resources effectively. To solve this challenge, we propose GPS, a HW/SW multi-GPU memory management technique that efficiently orchestrates inter-GPU communication using proactive data transfers. GPS offers the programmability advantage of multi-GPU shared memory with the performance of GPU-local memory. To enable this, GPS automatically tracks the data accesses performed by each GPU, maintains duplicate physical replicas of shared regions in each GPU's local memory, and pushes updates to the replicas in all consumer GPUs. GPS is compatible within the existing NVIDIA GPU memory consistency model but takes full advantage of its relaxed nature to deliver high performance. We evaluate GPS in the context of a 4-GPU system with varying interconnects and show that GPS achieves an average speedup of 3.0× relative to the performance of a single GPU, outperforming the next best available multi-GPU memory management technique by 2.3× on average. In a 16-GPU system, using a future PCIe 6.0 interconnect, we demonstrate a 7.9× average strong scaling speedup over single-GPU performance, capturing 80% of the available opportunity.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; *Peer-to-peer architectures.*

KEYWORDS

GPGPU, multi-GPU, strong scaling, GPU memory management, communication, heterogeneous systems

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480088

ACM Reference Format:
Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. 2021. GPS: A Global Publish-Subscribe Model for Multi-GPU Memory Management. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480088

1 INTRODUCTION

Graphics Processing Units (GPUs) are central to high-performance computing because of their high memory bandwidth and microarchitecture tailored for parallel execution. Because workload demands continue to grow beyond what single-GPU performance can provide, GPU manufacturers now offer systems comprising multiple GPUs to continue scaling application throughput [5, 40]. These aggregated multi-GPU systems provide teraflops of computational power and many terabytes per second of memory bandwidth [20, 42]. However, effectively managing these resources to extract performance from a multi-GPU system remains a challenge for many GPU developers.

One of the primary challenges to achieving scalable performance lies in managing the order of magnitude gap between local and remote memory bandwidths. If applications are naively partitioned across GPUs, most memory accesses will traverse (relatively) slow remote links, resulting in the inter-GPU interconnect becoming a performance bottleneck. Figure 1 demonstrates that for a variety of hard-to-scale HPC benchmarks on a 4-GPU system (see Section 6 for specifics), interconnect bandwidth limits scalability. A system with infinite interconnect bandwidth and a system with projected PCIe 6.0 performance attain 3× and 2× speedups over a single GPU, respectively, while using a current PCIe 3.0 interconnect can result in application performance 30% *slower* than its single-GPU counterpart.

The multi-GPU partitioning problem is difficult because the current techniques available for developers to effectively manage data in multi-GPU systems fall short. Unified Memory (UM) [21] provides a single memory address space accessible from any processor in the system by employing fault-based and/or hint-based page migration to move data pages to the local physical memory of the accessing processor. Although this enables UM to automatically migrate pages for locality, the page fault handling overheads are often performance prohibitive. Some programmers therefore resort to manual hints, but using hints effectively requires substantial tuning

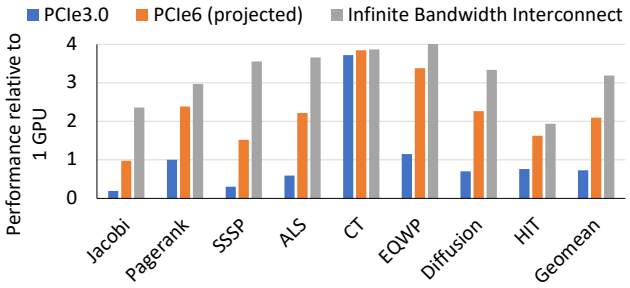

Figure 1: Many HPC programs strong-scale poorly due to insufficient inter-GPU bandwidth, as shown on a system with 4 NVIDIA GV100 GPUs.

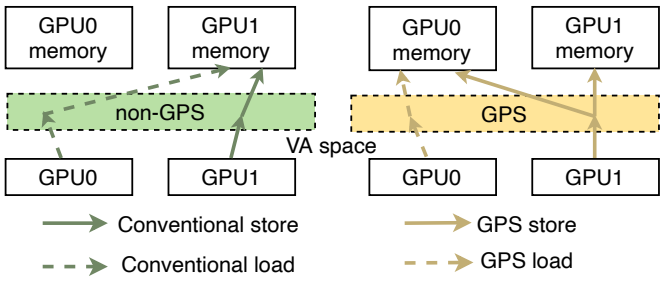

Figure 2: Load/store paths for conventional and GPS pages. Because GPS transfers data to consumers' memory proactively, all GPS loads can be performed to high bandwidth local memory.

effort on the part of the programmer. Peer-to-peer transfers [51], in which GPUs perform loads/stores directly to the physical memories of other GPUs, can potentially achieve good strong scaling when properly coordinated with computation. However, peer-to-peer loads perform remote accesses on demand, leading to compute stalls. Peer-to-peer stores avoid stalls, but can result in wastage of an already scarce interconnect bandwidth if sent to GPUs that do not require the pushed data. Frameworks such as Gunrock [60] and Groute [11] can improve strong scaling for some workloads but are typically domain-specific and limited in scope.

To overcome the limitations of these techniques, we propose GPS (GPU Publish-Subscribe), a multi-GPU memory management technique that transparently improves the performance of multi-GPU applications following a fully Unified Memory compatible programming model. GPS provides a set of architectural enhancements that automatically track which GPUs subscribe to shared memory pages, and through driver support, it replicates those pages locally on each subscribing GPU. GPS hardware then broadcasts stores proactively to all subscribers, enabling the subscribers to read that data from local memory at high bandwidth. Figure 2 shows the behavioral difference between GPS and conventional accesses. GPS loads always return from local memory, while GPS stores are broadcast to all subscribers. On the other hand, conventional load/stores result in local or remote accesses depending on physical memory location. GPS takes advantage of the fact that remote stores do not stall execution. Performing remote accesses on the write path instead of the read path hides latency and enables further optimizations to schedule and combine writes to use the interconnect bandwidth efficiently.

GPS successfully improves strong scaling performance because it can: (1) transfer data in a proactive, fine-grained fashion and overlap compute with communication, (2) issue all loads to local DRAM rather than to remote GPUs' memory over slower interconnects, and (3) perform aggressive coalescing optimizations that reduce inter-GPU bandwidth requirements without violating the GPU's memory model.

This work makes the following contributions:

- We propose and describe GPS, a novel HW/SW co-designed multi-GPU memory management system extension that leverages a publish-subscribe paradigm to improve multi-GPU system performance.
- We propose new simple and intuitive programming model extensions that can naturally integrate GPS into applications while conforming to the pre-existing NVIDIA GPU memory model.
- To minimize the utilization of scarce inter-GPU bandwidth, we propose a novel memory subscription management mechanism that tracks GPUs' access patterns and unsubscribes GPUs from pages they do not access.
- Evaluated in simulation with several interconnects against a 4 GPU system, GPS provides an average strong scaling performance of 3.0× over a single GPU capturing 93.7% of the available opportunity. In a 16 GPU system using a future PCIe 6.0 interconnect, GPS achieves an average 7.9× speedup over a single GPU and captures over 80% of the hypothetical performance of an infinite bandwidth interconnect.

2 BACKGROUND AND MOTIVATION

Proper orchestration of inter-GPU communication is essential for strong scaling in multi-GPU systems. With each new GPU and interconnect generation, the compute throughput, local memory bandwidth, and inter-GPU bandwidth increase. However, the bandwidth available to local GPU memory remains much higher than the bandwidth available to remote GPU memories. For example, Figure 3 shows that even though interconnect bandwidth has improved 38× while evolving from PCIe 3.0 [3] to NVIDIA's most recent NVLink and NVSwitch-based topology [38], it remains 3× slower than the local GPU memory bandwidth.

2.1 Inter-GPU communication mechanisms

Because local vs. remote bandwidth is a first-order performance concern, multi-GPU workloads typically use one of the following mechanisms (summarized in Figure 4) to manage data placement among the GPUs' physical memories.

- **Host-initiated DMA using cudaMemcpy**: `cudaMemcpy()` programs a GPU's DMA engine to copy data directly between GPUs and/or CPUs. Though CUDA provides the ability to pipeline compute and `cudaMemcpy()`-based transfers [55], implementing pipeline parallelism requires significant programmer effort and detailed knowledge of the applications' behavior in order to work effectively.
- **Fault-based migration via Unified Memory (UM)**: Unified Memory [21] provides a single unified virtual memory

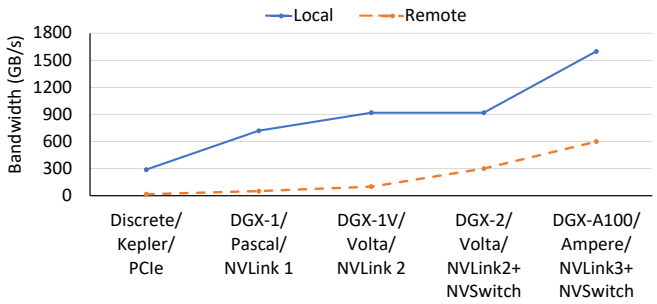

Figure 3: Local and remote bandwidths on varying GPU platforms. Despite significant increases in both metrics, a 3× bandwidth gap persists between local and remote memories.

address space accessible to any processor in a system. UM uses a page fault and migrate mechanism to perform data transfers among GPUs. Although this enables data movement among GPUs in an implicit and programmer-agnostic fashion, the performance overhead of these page faults typically is a first-order performance concern.

- **Hint-based migration via Unified Memory**: UM also offers the ability for programmers to provide hints to improve performance. Read-mostly, prefetching, placement, and "accessed-by" hints enable duplication of read-only pages across GPUs as well as page placement on specific GPUs and thus can reduce page faults if used correctly. However, pipelining prefetching and compute using hints to achieve fine-grained data sharing, our goal in GPS, is challenging even for expert programmers. Furthermore, crucially, UM does not support the replication of pages with at least one writer and multiple readers. Writes to read-duplicated pages "collapse" the page to a single GPU (usually the writer) and trigger an expensive TLB shootdown, thus degrading performance substantially [59]. GPS aims to provide a better solution for read-write pages accessed by multiple GPUs.
- **Peer-to-peer transfers**: GPU threads can also perform peer-to-peer loads and stores that directly access the physical memory of other GPUs without requiring page migration. In principle, peer-to-peer accesses can be performed at a fine granularity, overlap compute and communication, and incur low initiation overhead. However, peer-to-peer loads suffer from the high latency of traversing interconnects such as PCIe or NVLink, often stalling thread execution beyond the GPU's ability to mitigate those stalls via multi-threading. On the other hand, peer-to-peer stores typically do not stall GPU thread execution and can be used to proactively push data to future consumers without slowing down the computation phases of the application.
- **Message Passing Interface (MPI)**: MPI is a standardized and portable API for communicating via messages between distributed processes. With CUDA-aware MPI and optimizations such as GPUDirect Remote Direct Memory Access (RDMA) [45], the MPI library can send and receive GPU buffers directly, without having to first stage them in host memory. However, porting an application to MPI increases programmer burden, and it is hard to overlap compute and communication effectively by leveraging the pipelining features of MPI.

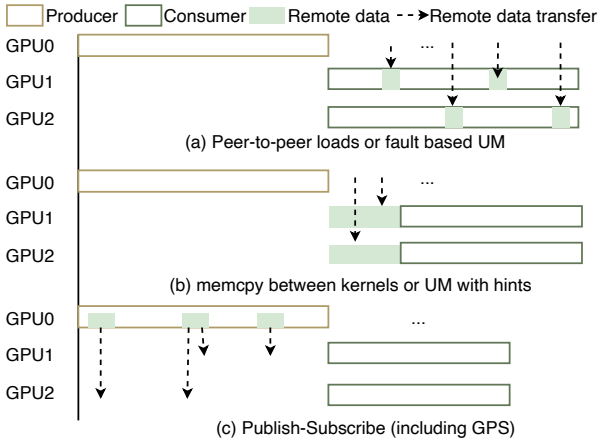

Figure 4: Data transfer patterns in different paradigms. In demand-based loads and UM, transfers happen on-demand; in memcpy, they happen bulk-synchronously at the end of producer kernel; in GPS, proactive fine-grained transfers are performed to all subscribers.

One way to avoid the remote access bottleneck is to transfer data from the producing GPUs to the consuming GPUs in advance, as soon as the data is generated. The consumers can then read the data directly from their local memory when needed. These proactive transfers help strong scaling for two reasons: (1) they provide more opportunities to overlap data transfers with computation, and (2) they improve locality and ensure that critical path loads enjoy higher local memory bandwidth. As such, GPS relies on proactive peer-to-peer stores to perform data transfers as the basis of its performance-scalable implementation.

2.2 Publish-subscribe frameworks

Although proactive fine-grained data movement can improve locality, performing broad all-to-all transfers wastes inter-GPU bandwidth in cases where only a subset of GPUs will consume the data. In these cases, tracking which GPUs read from a page and then transmitting data only to these consumers can save precious interconnect bandwidth. To track a page's consumers and propagate updates only to them, GPS adopts a conceptual publish-subscribe framework, which is often used in software distributed systems [7, 15, 18, 33].

Figure 5 shows a simple example of a publish-subscribe framework. It consists of publishers who generate data and subscribers who have requested specific updates. The publish-subscribe processing unit tracks subscription information at page granularity, receives data updates from publishers and forwards them to subscribers. This mechanism provides the advantage that publishers and subscribers can be decoupled, and subscription management is handled entirely by the publish-subscribe processing unit.

The major challenge faced by a publish-subscribe model relying on proactive remote stores is deciding which GPUs should receive the data and when the stores should be transmitted. We address this in Section 3.

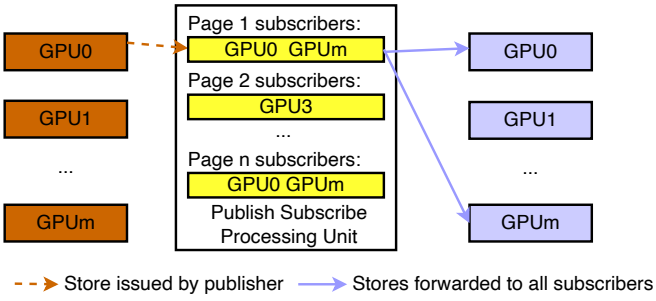

Figure 5: A simple publish-subscribe framework.

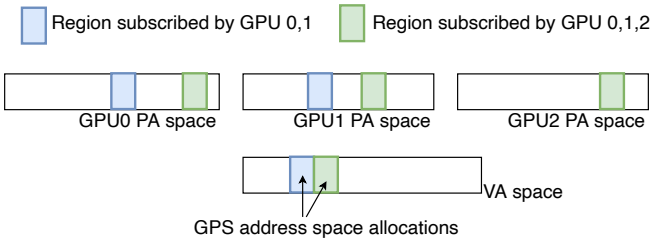

Figure 6: GPS address space: Allocations made are replicated in the physical memory of all subscribers.

2.3 GPU memory consistency

The NVIDIA GPU memory model [43] prescribes rules regarding the apparent ordering of GPU memory operations and the values that may be returned by each read operation. Of most relevance to GPS are the notions of *weak* vs. *strong* accesses, and the notion of *scope* associated with strong accesses. In short, `sys`-scoped memory accesses or fences are used to explicitly indicate inter-GPU synchronization. All other access types need not be made visible to or ordered with respect to memory accesses from other GPUs unless `sys`-scoped operations are used for synchronization. GPS makes use of these relaxed memory ordering guarantees to perform several hardware optimizations, as described later in Section 3.3, without violating the memory model.

3 GPS ARCHITECTURAL PRINCIPLES

In this section, we describe the different architectural principles upon which GPS is based. Several possible hardware implementations can support the GPS semantics with differing performance characteristics. These high-level design principles are intended to decouple the GPS concept from our specific proposed implementation. We highlight one particular implementation approach in Section 5.

3.1 The GPS address space

The *GPS address space*, which is an extension of the conventional multi-GPU shared virtual address space, enables programmers to incorporate GPS features into their applications through simple, intuitive APIs (described later in Section 4). As shown in Figure 6, all allocations made in the GPS address space have local replicas in all subscribing GPUs' physical memories. Subscription management is described in Section 3.2.

Loads and stores are issued to GPS pages in the same way as they are to normal pages, with the same syntax, although the underlying behavior is different. GPS' architectural enhancements intercept each store and forward a copy to each subscriber's local replica. Loads to GPS pages are also intercepted but are not duplicated. Instead, they are issued to the replica in the issuing GPU's local memory and can therefore be performed at full local bandwidth and without consuming interconnect bandwidth. In the corner case where the issuing GPU is not a subscriber to that page (e.g., because of an incorrect subscription hint), the load is issued remotely to one of the subscribers. This design offers GPS a significant advantage over existing multi-GPU programming frameworks: a programmer can integrate GPS into their workloads with only minor changes for allocation and subscription management and need not modify GPU kernels written for UM, except to apply performance tuning.

3.2 Subscription management

The key innovation behind GPS is a set of architectural enhancements designed to coordinate proactive inter-GPU communication for high performance. These architectural enhancements maintain subscription information to enable data transfer only to GPUs that require it, enabling local accesses during consumption.

GPS provides both manual and automatic mechanisms to manage page subscriptions. We describe the general mechanisms below, the APIs in Section 4, and the implementation in Section 5.

Manual subscription tracking: The manual mechanism allows the user to explicitly specify subscription information through subscription/unsubscription APIs that can be called while referencing each page (or a range of pages) in memory. Though this requires extra programmer effort, it can lead to significant bandwidth savings compared to an all-to-all subscription, even if the subscription list is not minimized.

Automatic subscription tracking: For applications with an iterative/phase behavior wherein the access patterns in each program segment match those of prior segments, the subscriber lists can be determined automatically. The repetition in iterative workloads enables GPS to discover the access patterns from one segment in an initial profiling phase and determine the set of subscriptions for subsequent execution.

Automatic subscriber tracking can be performed in one of two ways: subscribed-by-default, i.e., indiscriminate all-to-all subscription followed by an unsubscription phase, or unsubscribed-by-default, i.e., a GPU subscribes to a page only when it issues the first read request to that page. Either way, once captured, the sharer information then feeds into the subscription tracking mechanism used to orchestrate the inter-GPU communication.

It is important to note that the subscriptions are hints to GPS for both mechanisms and are not functional requirements for correct application execution. In other words, if a GPU issues a load to a page to which it is not a subscriber, it does not fault; the hardware simply issues the load remotely to one of the subscribers. Performing a peer-to-peer load to a remote GPU imposes a performance penalty but does not break functional correctness.

3.3 Functionally correct coalescing

While publish-subscribe models have been proposed in the past, a unique aspect of GPS is the way it exploits the weakness of the GPU memory consistency model described in Section 2.3. In particular, GPS performs aggressive coalescing of stores to GPS pages before those stores are forwarded to other GPUs, as described below.

Non-sys-scoped accesses: As described in Section 2.3, aside from sys-scoped operations, the GPU memory model only requires writes to be made visible to other GPUs upon execution of sys-scoped synchronization (e.g., memory fences). GPS uses this delayed visibility requirement to aggressively coalesce weak stores if their addresses fall within the same cache line. Stores need not be consecutive to be coalesced, as the GPU memory model allows store-store reordering as long as there is no synchronization or same-address relationship between the stores. This decreases the data transferred across the interconnect, saving valuable inter-GPU bandwidth. The coalescer must be flushed only upon sys-scoped synchronization, including the implicit release operation at the end of every grid.

The GPU memory model also does not require cache coherence between GPUs, i.e., a consistent store visibility order, unless stores are from the same GPU to the same address, have sys scope, or are synchronized by sys-scoped operations. While it is possible for GPS stores broadcast from different GPUs to the same address to cross each other in flight and therefore arrive at different consumer GPUs in different orders, this is *not* a violation of the memory model. Weak stores performed by different GPUs to the same address simultaneously without synchronization are racy, and therefore, no ordering guarantees need to be maintained. As long as proper synchronization is being used, weak writes will only be sent to a particular address from one GPU at a time, and point-to-point ordering will ensure that all consumer GPUs see those stores arriving in the same order. In this way, GPS maintains all of the required memory ordering behavior.

sys-scoped accesses: These writes are intended for synchronization and must be kept coherent across all GPUs. Therefore, GPS neither coalesces nor broadcasts such writes but instead simply handles them as traditional GPUs do. Specifically, all sys-scoped accesses are sent to a single point of coherence and performed there. Typically, the number of sys-scoped operations in programs is low, as they are only used when grids launched concurrently on multiple GPUs need to explicitly synchronize through memory. Hence, the cost of not coalescing system-scoped strong stores is minimal. Further discussion of the handling of sys-scoped writes in our GPS implementation is described in Section 5.3.

The design choices described above ensure that GPS can deliver consistent performance gains without breaking backward compatibility with the GPU programming model or memory consistency model. This compatibility enables developers to easily integrate GPS into their applications with minimal code or conceptual change.

4 GPS PROGRAMMING INTERFACE

We next describe the programming interface that an application developer uses to leverage GPS features. We seek to develop a minimal, simple programming interface to ease the integration of GPS into existing multi-GPU applications. GPS API functions are implemented in the CUDA library and GPU driver, just as the existing CUDA APIs are. Listing 1 shows sample code.

```
__global__ void mvmul(float* invec, float* outvec, ...) {
  ...
  // Stores to outvec in the GPS address space are
       forwarded to the replicas at each subscriber GPU
  for(int i=0; i<mat_dim; i++)
   outvec[tid] += mat[tid*mat_dim+i] * invec[i];
}
int main(...) {
 // enable GPS for mat, vec1, and vec2
 cudaMallocGPS(&mat, mat_dim*mat_dim_size);
 cudaMallocGPS(&vec1, mat_dim_size);
 cudaMallocGPS(&vec2, mat_dim_size);
 cudaMemset(vec2, 0, mat_dim_size);
 for(int iter=0; iter<MAX_ITER; iter++) {
  // Automatic profiling: all GPUs are tentatively
       subscribed to all GPS pages at the start
  if (iter==0) cuGPSTrackingStart();
  for(int device=0; device<num_devices; device++) {
   cudaSetDevice(device);
   mvmul<<<num_blocks, num_threads, stream[device]>>>(
       mat, vec1, vec2, ...);
   mvmul<<<num_blocks, num_threads, stream[device]>>>(
       mat, vec2, vec1, ...);
  }
  // GPUs are unsubscribed from pages they did not touch
  if (iter==0) cuGPSTrackingStop();
}
```

Listing 1: A sample GPS application. GPS requires code changes only for GPS allocation and tracking as highlighted in yellow.

Memory allocation and release: GPS provides an API call, cudaMallocGPS(), as a drop-in replacement for cudaMalloc() (for GPU-pinned memory) or cudaMallocManaged() (for Unified Memory) APIs. This call allocates memory in the GPS virtual address space and backs it with physical memory in at least one GPU. At allocation, the programmer can pass an optional manual parameter to indicate that subscriptions will be managed explicitly for the region. Otherwise, GPS performs automatic subscription management. GPS re-purposes the existing cudaFree() function to release a GPS memory region.

Manual subscription: To allow expert programmers to explicitly manage subscriptions, GPS overloads the existing cuMemAdvise() API used for providing hints to UM with two additional hints to perform manual subscription and unsubscription. Specifically, GPS uses new flag enums CU_MEM_ADVISE_GPS_SUBSCRIBE and CU_MEM_ADVISE_GPS_UNSUBSCRIBE for subscription and unsubscription, respectively. Upon subscription, GPS backs the region with physical memory on the specified GPU. When a programmer unsubscribes a GPU from a GPS region, GPS removes the GPU from the set of subscribers for that region and frees the corresponding physical memory. GPS ensures that there is at least one subscriber to a GPS region and will return an error on attempts to unsubscribe the last subscriber, leaving the allocation in place.

Automatic subscription and profiling phase: As described in Section 3.2, the automatic subscription mechanism comprises a hardware profiling phase during which the mechanism observes application access patterns and determines a set of subscriptions. This profiling phase requires the user to demarcate the start and end of the profiling period using two new APIs, cuGPSTrackingStart()

and `cuGPSTrackingStop()`, which are similar to the existing CUDA calls `cuProfilerStart()` and `cuProfilerStop()`. GPS automatically updates subscriptions at the end of the profiling phase; a GPU remains a subscriber if and only if it accessed the page during profiling. Thus, upon receiving `cuGPSTrackingStop()`, GPS invokes the API `cuMemAdvise(..., CU_MEM_ADVISE_GPS_UNSUBSCRIBE)` to unsubscribe GPUs from any page they did not access during profiling. (Recall that a GPU may still access a page to which it is not subscribed, but such accesses will be performed remotely at reduced performance; hence, profiling need not be exact to maintain correctness).

5 ARCHITECTURAL SUPPORT FOR GPS

We now describe one possible GPS hardware implementation that extends a generic GPU design, such as NVIDIA's recent Ampere GPUs. Our hardware proposal comprises two major components. First, it requires one bit in the GPU page table entry (PTE), the GPS bit, to indicate whether a virtual memory page is a GPS page (i.e., potentially replicated). Second, it requires a new hardware unit to propagate writes to GPUs that have subscribed to particular pages.

5.1 GPS memory operations

GPS must support the following basic memory operations:

Conventional loads, stores, and atomics: Memory accesses to non-GPS pages (virtual addresses for which the GPS bit is not set in the PTE) proceed as they do on conventional GPUs, through the existing GPU TLB and cache hierarchy to either local or remote physical addresses.

GPS loads: Figure 7 shows the paths taken by writes and reads to GPS pages. Note that this figure is a simplified view intended to highlight only the modifications required by GPS. For loads issued to the GPS address space by a GPU that is a subscriber to the page, the conventional GPU page table is configured at the time of subscription to translate the virtual address to the physical address of the local replica. GPS loads thus follow the same path as conventional loads to local memory, as shown by R1, R2, R3 in Figure 7. In the uncommon case, if a GPU is not subscribed to this particular page, either the load forwards a value from the remote write queue (Section 5.2) if there is a hit or it issues remotely to one of the subscribers.

GPS stores and atomics: Stores to GPS pages initially proceed as normal stores, as shown by W1 and W2 in Figure 7. When a thread issues a store to an address whose GPS bit is set in the conventional TLB, and for which there is a local replica, the write operation is forwarded to the local replica with both virtual and physical address (W3), ensuring that subsequent local reads from the same GPU thread will observe the new write, a requirement of the existing GPU memory model. This pattern also ensures that the L2 cache holding the local replica will serve as a common intra-GPU ordering point for stores to that address prior to their being forwarded outside of the GPU, as the memory model requires. In the uncommon case, there is no local replica, and a dummy physical address is used. In addition, whether or not there is a local replica, the write is also forwarded with its virtual address to the GPS unit (described next) for replication to remote subscribers (W4, W5, W6). Atomics follow the same behavior as stores.

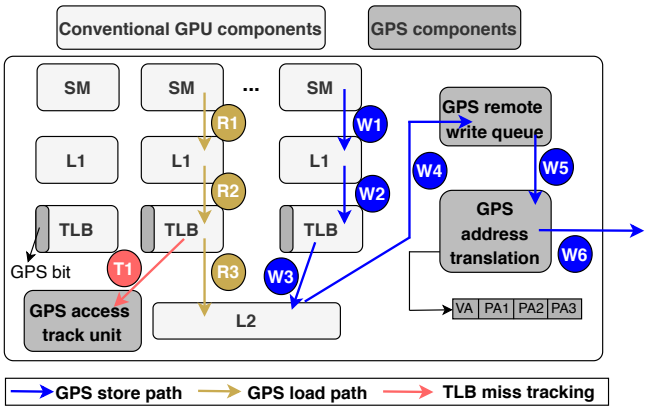

Figure 7: Modifications to GPU hardware needed for GPS provisioning.

5.2 GPS hardware units and extensions

Page table support: Our GPS implementation modifies the baseline GPU page table entries (PTEs) to re-purpose a single currently unused bit (of which there are many [37]) to indicate whether a given page is a GPS page. When this bit is set, the virtual address is GPS-enabled, and the stores to the page will be propagated to the GPS units described below. The baseline GPU virtual memory system remains unchanged aside from this extension.

Our GPS support also introduces a new secondary page table, the GPS page table, for tracking multiple physical mappings that can coexist for a given virtual address when multiple GPUs subscribe to a page. This limited page table for the GPS address space is a variant of a traditional 5-level hierarchical page table with very wide leaf PTEs. Notably, the GPS page table lies off the critical path for memory operations, as it is used only for remote writes triggered by writes to GPS pages. These remote writes are already aggressively coalesced, so the additional latency of the GPS address translation unit can overlap with the coalescing period and adds only a small additional latency. Furthermore, by design, these writes are not required to become visible until the next synchronization boundary, so they are not latency-sensitive. Therefore, the latencies imposed by the address translation unit are not a critical factor, even under TLB misses.

Each GPS-PTE contains the physical page addresses of all the remote subscribers to that page, as shown in Figure 7. The GPS-PTE is sized at GPU initialization based on the number of GPUs in the system. For example, with 64KB pages, for a Virtual Page Number (VPN) size of 33 bits and Physical Page Number (PPN) size of 31 bits [37], for a 4 GPU system, the minimum GPS-PTE entry size is 126 bits.

We choose to allocate memory in the GPS address space using 64KB pages for two reasons. First, the negative impact of false sharing due to large pages is multiplied due to GPS' replication of remote stores. Second, the GPU's conventional TLB is already sized to provide full coverage of the entire VA range [47]. As discussed later in Section 7.4, the GPS address translation unit requires a surprisingly small translation capacity even with 64KB pages. Therefore, neither TLB lies on the execution's critical path.

Coalescing remote writes: If a store's translation information indicates it is to a GPS-enabled address, the GPS write proceeds to the remote write queue (W4). This queue is fully associative and virtually addressed at cache block granularity. The queue coalesces all writes to the same cache block (unless they are `sys`-scoped) and buffers them until they are drained to remote GPU memory. This simple optimization results in substantial inter-GPU bandwidth savings for many applications yet maintains weak memory model compatibility.

Whenever the occupancy of the write combining buffer reaches a high watermark, it seeks to drain the least recently added entry to the remote destinations. In our evaluation, we set this high watermark to one less than the buffer's capacity to maximize coalescing opportunity. On draining, the flushed entry moves to the GPS address translation unit. Additionally, the remote write queue unit must fully drain at synchronization points, e.g., when a `sys`-scoped memory fence is issued. In our final proposal with 512 entries, the GPS-write buffer requires 68 KB of SRAM storage, an insignificant chip area for a single resource shared among all SMs in the GPU.

GPS address translation: When a GPS store reaches the GPS address translation unit (W5), it looks up translation information cached from the GPS page table in its internal, wide GPS-TLB. Much like conventional TLB misses, GPS-TLB misses trigger a hardware page walk that fetches a GPS-PTE entry containing the physical addresses for all subscribers. Once translated, the GPS address translation unit sends a duplicate copy of the store for each subscriber to the inter-GPU interconnect (W6). The GPS address translation unit also drains at synchronization points.

Access tracking unit: Our GPS implementation uses subscribed-by-default profiling: it employs a dynamic mechanism that typically begins by subscribing all GPUs to the entirety of GPS allocations at the beginning of execution. As discussed in Section 4, GPS tracks the access patterns during a profiling phase, after which it unsubscribes each GPU from pages they did not access. Although the early over-subscription initially transfers more data than is required, unsubscribed-by-default profiling incurs stalls due to faults or remote accesses on the first touch and hence is more expensive.

The GPS access tracking unit provides the hardware support for runtime subscription profiling. It maintains a bitmap in DRAM with one bit per page in the GPS address space. Misses at the last level conventional GPU TLBs to pages in the GPS virtual address space are forwarded to the access tracking unit, which sets the bit corresponding to the page (as shown by T1 in Figure 7). TLB misses are infrequent yet cover all pages accessed by the GPU, so the bandwidth required to maintain this bitmap is low (typically only 1.4 TLB misses per thousand cycles [47]). We maintain the bitmap in DRAM since the bitmap is used only during the initial profiling and unsubscription. Tracking a 32GB virtual address range, the bitmap requires only 64KB of DRAM, and updates can be aggressively cached or write-combined to minimize DRAM bandwidth impact. Thus, the total area and energy consumed by these hardware extensions are negligible relative to the GPU SoC.

The bitmap managed by the access tracking unit is read by the GPU driver during the `cuGPUTrackingStop()` API call and used to configure the conventional and GPS page tables appropriately. GPS pages with only a single subscriber are downgraded to conventional pages within the page tables, as duplication of writes to such pages at the conventional TLB (Section 5.1) is an unnecessary waste of resources when there is only one subscriber. For GPS pages with multiple subscribers, the GPS bit is set in the conventional page table entry for that page, and the GPS page table entry for the page is updated to record the physical addresses of all subscribers' replicas for that page.

5.3 Discussion

Coalescing in the L2 cache: An alternative implementation strategy for GPS would be to perform coalescing directly within the L2 cache rather than provisioning a dedicated structure. Given that the GPS remote write queue amounts to only a few kilobytes of state and the L2 size is in megabytes (6MB in NVIDIA V100 GPUs and 40MB in NVIDIA A100 GPUs [39]), cache capacity impact would be negligible. We chose to implement a dedicated write queue to isolate its impact from any other cache interference effects and so that the only change required to the L2 cache will be a shim at the ingress point, which forwards GPS writes to the GPS remote write queue.

GPS remote write queue addressing: Our GPS implementation assumes the remote write queue is virtually addressed, irrespective of whether it is maintained as a dedicated SRAM structure or as specially marked and reserved cache lines in L2. If GPS were to perform translation prior to the GPS write queue, a remote store would require one entry per subscribing GPU (one per remote physical address); performing translation as the stores are drained to the interconnect conserves space in the write queue.

Handling sys-scoped writes: Strong `sys`-scoped writes must be kept coherent across all GPUs. Our GPS implementation handles `sys`-scoped writes in the same way that UM handles writes to pages annotated with read-mostly `cudaMemAdvise` hints. Upon detection of a `sys`-scoped store to a GPS page, the access faults, all in-flight accesses to the page are flushed, and the page is collapsed to a single copy and demoted to a conventional page (i.e., its GPS bit is cleared). Accesses to the page are all issued to the GPU hosting the single physical copy from this point on. This approach ensures coherence and same-address ordering for this access and all future accesses to the page.

As mentioned in Section 3.3, `sys`-scoped accesses are rare, and hence the impact on typical programs is minimal. The user is expected to explicitly opt pages holding synchronization variables out of GPS (i.e., use `cudaMalloc` instead of `cudaMallocGPS`). If the user provides incorrect hints, then just as with UM, there will be a performance penalty. Nevertheless, the execution remains functionally correct.

Handling memory oversubscription: If the GPU driver swaps out a page from a subscriber due to oversubscription, that GPU will be unsubscribed and will access that page remotely.

6 EXPERIMENTAL METHODOLOGY

To evaluate the benefits of GPS, we extend the NVIDIA Architectural Simulator (NVAS) [57] to model multi-GPU systems comprising NVIDIA GV100 GPUs on PCIe, with parameters shown in Table 1. The simulator is driven by application traces collected at the SASS (GPU assembly) level using the binary instrumentation

GPU Parameters	
Cache block size	128 bytes
Global memory	16 GB
Streaming multiprocessors (SM)	80
CUDA cores/SM	64
L2 Cache size	6 MB
Warp size	32
Maximum threads per SM	2048
Maximum threads per CTA	1024
GPS Structures	
Remote write queue	512 entries
Remote write queue entry size	135 bytes
TLB	8-way set associative
TLB size	32 entries
Virtual address	49 bits
Physical address	47 bits

Table 1: Simulation settings, based on NVIDIA V100.

tool NVBit [58] on real hardware. These traces contain CUDA API events, GPU kernel instructions, and memory addresses accessed, but no pre-recorded timing events. The simulator models the timing aspects of the trace replay in accordance with the GPU and interconnect architectural models and respects all functional dependencies such as work scheduling, barrier synchronization, and load dependencies. We have specifically calibrated the link and switch parameters in our interconnect models to match several (sometimes speculative) PCIe generations. This simulator has been correlated across a wide range of benchmarks and GPU models but remains fast enough to model complex multi-GPU systems and the hard-to-scale applications suitable for evaluating GPS.

We evaluate a suite of multi-GPU applications shown in Table 2. These include all applications used to evaluate PROACT [34]. We also study those applications from the Tartan benchmark suite [29], whose strong scaling performance was bound by inter-GPU communication when measured on real systems. These applications also possess varying communication patterns, giving us a broader opportunity to evaluate GPS. For the Tartan applications not bound by inter-GPU communication, we found that GPS obtains the same performance as the native version and have not included them in the interest of space. We modify the applications only to implement the different multi-GPU programming paradigms, and the partitioning of applications across multi-GPUs remains the same as the original code for all paradigms. All our application variants are written in CUDA and compiled using CUDA 10.

To demonstrate the ability of GPS to improve multi-GPU scalability, we compare it with several contemporary multi-GPU programming paradigms as discussed in Section 2.1:

Unified Memory without Hints: We simulate baseline Unified Memory without user-provided hints. Application code allocates shared memory regions using `cudaMallocManaged()` API. By default, the simulator allocates pages on the first GPU that touches the page. Subsequent accesses by peer GPUs to the same page will result in fault-based page migration as described in Section 2.1.

Unified Memory with Hints: For this paradigm, we hand-tune each application using a combination of four manual hints, namely read mostly, accessed by, prefetch, and preferred location hints. Based on the compute partitioning across GPUs, we set the GPU that issues writes to a given memory region as its *preferred location*. The most proactive approach we can configure with UM hints is to pick one consumer to be the preferred location, and since each producer of a page is always also a consumer of the page in our applications, that was a convenient and close-to-optimal choice. We also set GPUs that read from remote pages as *accessed by* those GPUs. Although *read-mostly* hints are generally effective, we did not use them because our applications had no read-only pages accessed by multiple GPUs. Before each kernel launch, we enable GPUs to prefetch remote regions they may access through prefetch hints.

Remote Demand Loads (RDL): While GPS performs all loads locally by issuing the stores to all subscribers, RDL performs the converse: it issues stores to local memory and loads to the most recent GPU to issue a store to a given page. We believe that this paradigm is representative of an expert programmer who manually tracks writers to each page. We simulate this expertise by explicitly tracking the latest write to each page in the simulator and using this information during address translation to issue the read to the appropriate GPU.

Memcpy: This paradigm duplicates data structures among all GPUs and broadcasts updates via `cudaMemcpy()` calls at the synchronization barriers. This duplication ensures that all data structures are resident in local GPU memory when accessed by kernels in the subsequent synchronization phase; there are no remote accesses during kernel execution. However, there is also no overlap between data transfers and compute.

GPS with automatic subscription: We implement GPS with automatic subscription management by modifying applications as described in Section 4 and marking all memory allocations as GPS allocations.

Infinite bandwidth: Finally, we provide an *infinite bandwidth* comparison, which establishes an upper bound on achievable multi-GPU performance if all data were always accessible locally at each GPU (i.e., it ignores all transfer costs). We obtain this comparison by eliding the data transfer time from the memcpy variant.

7 EXPERIMENTAL RESULTS

GPS relies on fine-grained, proactive data transfers to remote GPUs during kernel execution to optimize GPU locality. The subscription management mechanism ensures that only the required data is transferred, resulting in interconnect bandwidth savings. GPS performance benefits arise for three reasons: (1) GPS proactively publishes updates to subscribers, enabling them to fetch hot-path data from high bandwidth local memory. (2) By automatically identifying subscribers for a given page, GPS publishes updates only to the GPUs that require them, resulting in significant interconnect bandwidth savings. (3) Coalescing in the GPS write queue results in substantial bandwidth reduction, especially for applications where subscriptions alone are not sufficient to achieve peak performance.

7.1 End-to-end performance

Figure 8 shows the 4-GPU application speedup over a single GPU for the different programming paradigms described in Section 6.

Application	Description	Predominant Communication Pattern
Jacobi	Iterative algorithm that solves a diagonally dominant system of linear equations	Peer-to-peer
Pagerank	Algorithm used by Google Search to rank web pages in their search engine results	Peer-to-Peer
SSSP	Shortest path computation between every pair of vertices in a graph	Many-to-many
ALS	Matrix factorization algorithm	All-to-all
CT	Model Based Iterative Reconstruction algorithm used in CT imaging	All-to-all
B2rEqwp	3D earthquake wave-propogation model simulation using 4-order finite difference method	Peer-to-peer
Diffusion	A multi-GPU implementation of 3D Heat Equation and inviscid Burgers' Equation	Peer-to-peer
Hit	Simulating Homogeneous Isotropic Turbulence by solving Navier-Stokes equations in 3D	Peer-to-peer

Table 2: Applications under study.

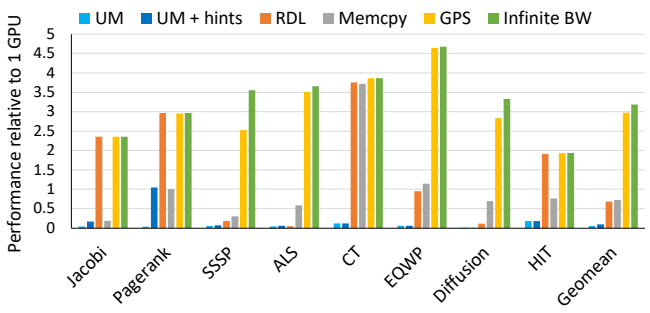

Figure 8: 4-GPU speedup of different paradigms.

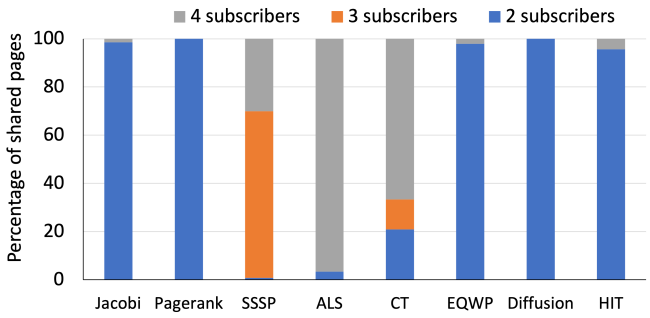

Figure 9: Subscriber distribution for shared application pages. GPS subscriptions result in interconnect bandwidth savings for all pages with less than 4 subscribers.

First, we observe that the unified memory paradigm is ineffective for these applications, despite having attractive programmability properties, due to the cost of page faults necessary to migrate data between GPUs. While work is ongoing [9, 64], it is unclear if executing page faults on the GPU's critical load path in multi-GPU systems will ever scale well.

Second, though applying UM hints results in performance benefits over baseline UM, we found that extracting maximum benefits out of hints by achieving fine-grained data sharing requires detailed knowledge about the data access patterns of the applications along with significant programmer effort. The performance penalty for incorrectly applied hints is still substantial, ranging from slow remote access in the best case to thrashing page migrations and expensive faults and TLB shootdowns in the worst case. Also, writes to pages replicated across subscribers result in the replicated pages collapsing back to the writer, further degrading performance. This degradation can be somewhat mitigated through profiling and fine-tuning of hints, but generally this approach is only effective after substantial expert effort and nevertheless faces fundamental limitations like the inability to replicate read-write pages, giving GPS an advantage.

Third, memcpy at kernel boundaries performs well for CT, but on average does not achieve any improvement over a well-optimized single GPU implementation, due to inefficient interconnect utilization during compute phases. This finding is significant because it demonstrates that though using programmer-directed memcpy to optimize locality is the most common programming technique today, it is unlikely to result in good strong scaling performance for applications bound by inter-GPU communication.

Finally, remote data loading performs well for applications where multi-threading is sufficient to hide remote load latencies; however, for others, these loads lie in the critical path and can have a severe adverse effect on performance.

The GPS approach ensures both a good overlap of communication and computation phases while at the same time decreasing the unwanted stores to GPUs that will not access them (locally or remotely). As a result, GPS offers an overall average speedup of 3× (out of 3.2× possible) over the single-GPU programs. Furthermore, we observe that EQWP achieves greater than 4× speedup due to an improvement in L2 hit rate from 55% to 68% when scaling to 4 GPUs due to the increase in aggregate cache capacity.

7.2 Benefits of subscription tracking

Figure 9 shows the distribution of GPS pages with more than one subscriber at the beginning of the GPS execution phase. Whereas some applications do perform all-to-all transfers for nearly all pages in the GPS space (ALS), other applications (like Jacobi) require only one remote subscriber for most pages because of how the algorithm performs boundary exchange of halos. The variation in these applications' subscription sets supports the idea that programmer efficiency is maximized by allowing promiscuous subscription to the GPS space, with automatic hardware unsubscription when the programmer cannot easily determine the ideal subscription set for each GPU.

Figure 10 compares the total data moved over the interconnect for all the paradigms. We normalize the numbers for all the paradigms to the memcpy variant since it copies all shared data exactly once

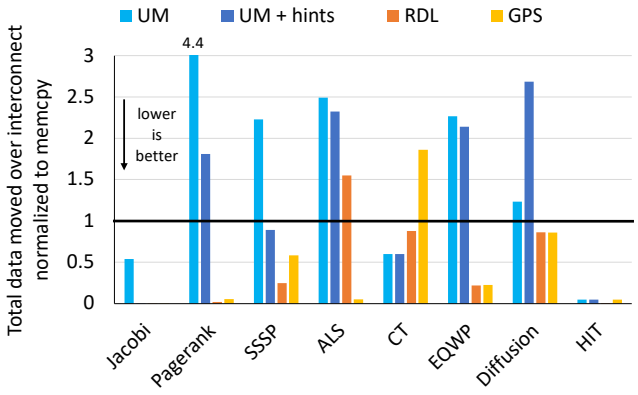

Figure 10: Total data moved over interconnect normalized to memcpy (bulk-synchronous transfers).

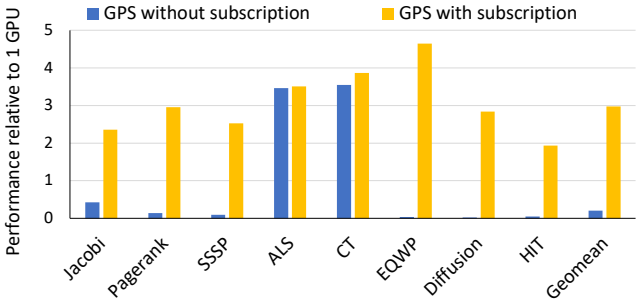

Figure 11: Performance sensitivity to subscription.

across all the GPUs, thus enabling us to quantify the additional write traffic introduced by each paradigm. The figure shows that the effectiveness of each technique is highly dependent on the access pattern of the applications.

We find that Unified Memory often causes significant increases in interconnect traffic compared to manual memcpy because the presence of multiple subscribers to the same page results in pages thrashing back and forth between the multiple GPUs that access it. The exceptions are Jacobi and CT, where the interconnect traffic due to memcpy needlessly copying data to GPUs that do not access them outweighs the traffic due to page migrations. Adding hints to UM decreases the total data moved when compared to UM in all cases except diffusion, where more fine-grained prefetching hints are required to avoid over-fetching pages needlessly. RDL moves less data than memcpy in all cases except for ALS, wherein a lack of temporal locality results in the same cacheline being fetched multiple times over the interconnect.

Compared to other paradigms, GPS's unsubscription mechanism drastically reduces the total data transferred over the interconnect in most cases. However, compared to the memcpy paradigm, there may be improvement or degradation in the amount of data transferred. We observe improvements when GPS allows small granularity updates to occur within a page, in contrast to bulk DMA

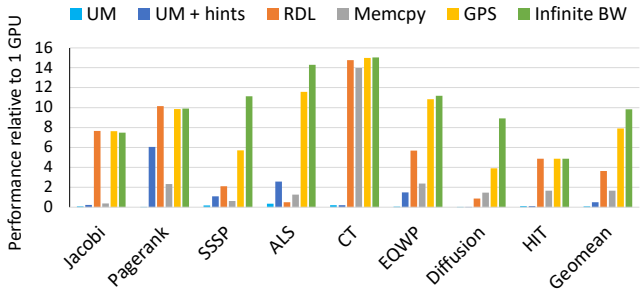

Figure 12: 16-GPU performance achieved by different paradigms.

transfers. When degradation occurs, it is typically due to the GPS write combining buffer failing to coalesce multiple writes to the same block effectively. However, if the excess writes do not saturate the interconnect, they will not typically stall GPU execution.

The end-to-end performance benefits of subscription tracking are shown in Figure 11, which compares the performance of GPS with and without subscription. The bandwidth savings due to subscription tracking is the primary factor in GPS's scalable performance results for most applications. The exceptions are ALS and CT, where a majority of the shared pages are subscribed by all the GPUs, resulting in significant all-to-all transfers.

7.3 Scalability beyond 4-GPUs

To understand GPS's scalability as GPU counts in a system increase, we simulate the performance of the different programming paradigms on a system comprising 16 NVIDIA GV100 GPUs using a projected PCIe 6.0 interconnect (operating at 128GB/s). Figure 12 shows the performance relative to one GPU for each of these paradigms. The application performance trend across the five techniques is similar to the 4-GPU trends discussed in Section 7.1. While current paradigms do not scale well on average, GPS achieves scalable performance with a mean speedup of 7.9×, capturing 80% of the performance opportunity available when modeling an infinite bandwidth interconnect.

7.4 Sensitivity Studies

Interconnect bandwidth: Figure 13 provides the geometric mean performance of each programming paradigm shown in Figure 8 while increasing the throughput of the inter-GPU interconnect. Our results show that despite expected dramatic improvements in PCIe and other inter-GPU interconnects [4, 41], strong scaling remains difficult to achieve using traditional GPU multi-programming paradigms. Conversely, as GPUs move to higher performance interconnects, GPS will approach the limits of performance scalability by making efficient use of inter-GPU bandwidth.

Write queue size: Sizing the GPS remote write queue properly is critical to overall GPS system performance. An ideal queue contains enough entries to exploit applications' temporal locality but is not so large that the associative lookup operations become overly expensive. Figure 14 studies the sensitivity of buffer size versus achieved hit rate. With 512 buffer entries all applications achieve

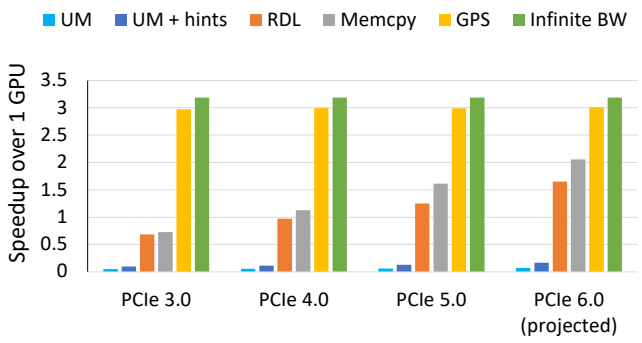

Figure 13: Sensitivity to interconnect bandwidth.

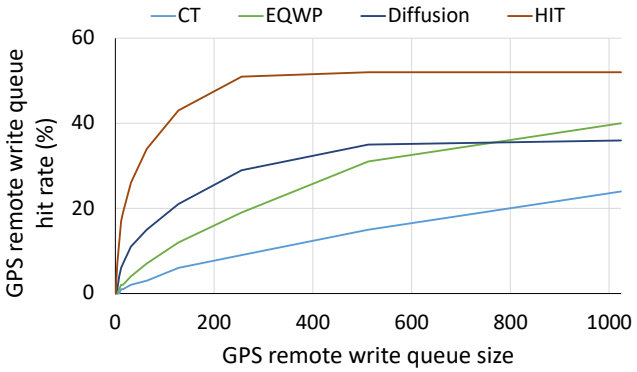

Figure 14: Sensitivity to GPS write queue size.

near peak performance. Further performance is difficult to capture due to random writes that have neither temporal nor spatial locality. We note that Jacobi exhibits a 0% hit rate since all spatial locality is fully captured in the coalescer internal to the SM, while Pagerank, ALS, and SSSP exhibit a 0% hit rate since they predominantly issue atomic operations whose coalescing is not supported by the GPS write queue.

GPS-TLB Size: In virtual memory systems, the translation information for a virtual address will typically be fetched only once (upon the first request to the page), and subsequent accesses to the page will result in TLB hits. The GPS-TLB is no exception and it is important to size the GPS-TLB appropriately to capture locality in the GPS space. GPUs now have thousands of entries in their last level TLB [35] and we initially expected the GPS-TLB to require a similar number of entries. However, we found that the GPS-TLB hit rate approaches 100% at just 32 entries. We find that because the GPS-TLB only services a fraction of the entire GPU memory space (GPS-allocated heap pages) and it does not service read requests to the GPS address space (they are typically serviced from the normal local memory system), the GPS-TLB is under substantially less pressure than the general-purpose GPU TLBs, and can thus be much smaller.

Page size: To measure the impact of the virtual memory page size, we study the performance of GPS under three page sizes, namely, 4kB, 64kB, and 2MB. While a smaller page size can decrease the false sharing of GPS pages, it significantly increases the pressure on all the TLBs in the GPU, resulting in the 4kB variant being 42% slower than 64kB. On the contrary, though a larger page size enables improved TLB hit rates, the interconnect traffic increases due to larger number of redundant remote transfers, resulting in the 2MB variant being 15% slower than 64kB. Therefore, 64kB is the sweet spot we focus on in our evaluation.

7.5 Limitations of the GPS approach

The GPS paradigm does not achieve maximum theoretical performance due to three primary issues. First, even though a GPU may subscribe to a page, it may only require updates to a portion of the (64kB) page, yet the interconnect will still transmit all updates to this page due to false sharing. Second, GPUs typically issue interconnect transfers at cache line granularity. We observe that not all bytes within a cache block are updated for many HPC applications, resulting in the transfer of some unneeded bytes. Third, although the write combining buffer coalesces stores to the same cache block, if the stores are temporally distant, the prior store might have already been flushed from the write combining buffer before the subsequent store to the same cache block arrives. Despite these inefficiencies, GPS represents a significant performance improvement over the state-of-the-art in multi-GPU performance scalability while also providing a simple and universal programming interface.

8 RELATED WORK

Prior work [6, 9, 11, 25, 32, 62] has explored the use of various hardware and software mechanisms to improve multi-GPU performance. Our work explores employing the publish-subscribe paradigm for strong scaling in multi-GPU systems. Prior work [49, 50, 56] has also proposed different mechanisms for inter-GPU coherence. GPS does not implement an expensive coherence protocol because it is not required for GPU memory model compatibility.

Several works propose multi-GPU memory management solutions. Although Griffin [9] optimizes page migration, reads to pages accessed by more than one GPU still result in on-demand remote reads that can fall on the execution critical path. GPS avoids these demand reads. CARVE [62] derives its benefits from caching remote data in local DRAM. However, it does not proactively update locally cached data, resulting in demand reads to data updated by peer GPUs. Only subsequent reads, after the first demand miss, are serviced from the DRAM cache, thus benefiting only workloads with substantial data re-use.

Several teams have studied prefetching in the context of single GPUs [26, 27, 52], but multi-GPU systems pose new challenges due to concurrent accesses and the high cost of GPU TLB shootdowns to migrate pages among the GPUs. NVIDIA's Unified Memory allows programmers to explicitly provide data placement hints [36] to improve the locality through programmer-controlled prefetching and decrease the rate of page migration in multi-GPU systems. It also provides a mechanism to allow read-only page duplication among GPUs, but upon any write to the page, it must collapse back to a single GPU. In future work, it could be possible to improve UM performance by layering it on top of a GPS-like system.

The publish-subscribe communication paradigm for distributed interaction has been explored by prior work [2, 7, 8, 17]. Hill et al. [23] propose a Check-In/Check-out model for shared-memory machines. The more traditional alternative to publish-subscribe support is NUMA memory management. Dashti et al. [14] develop and implement a memory placement algorithm in Linux to address traffic congestion in modern NUMA systems. Many other works [1, 16, 28, 48, 63] perform NUMA-aware optimizations to improve performance, and hardware-based peer caching has been explored but is yet to be adopted by GPU vendors [10, 13, 30, 46, 54]. Recently DRAM-caches for multi-node systems [12] have been proposed to achieve large capacity advantages.

Prior work has also explored scoped synchronization for memory models [19, 22, 24, 31, 44, 61]. Non-scoped GPU memory models are simpler [53], but do not permit the same type of coalescing as GPS, which makes explicit use of scopes.

9 CONCLUSION

Strong scaling in multi-GPU systems is a challenging task. In this work, we proposed and evaluated GPS, a HW/SW multi-GPU memory management technique to improve strong scaling in multi-GPU systems. GPS automatically tracks the subscribers to each page of memory and proactively broadcasts fine-grained stores to these subscribers. This enables each subscriber to read data from their local memory at high bandwidth. GPS provides significant performance improvement while retaining compatibility with conventional GPU programming and memory models. Evaluated on a model of 4 NVIDIA V100 GPUs and several interconnect architectures, GPS offers an average speedup of 3.0× over 1 GPU and performs 2.3× better than the next best available multi-GPU programming paradigm. On a similar 16 GPU system, GPS captures 80% of the available performance opportunity, a significant lead over today's current multi-GPU programming models.

ACKNOWLEDGMENTS

The authors thank Zi Yan and Oreste Villa from NVIDIA Research for their support with NVAS and the anonymous reviewers for their valuable feedback. This work was supported by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

REFERENCES

[1] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[2] Marcos K Aguilera, Robert E Strom, Daniel C Sturman, Mark Astley, and Tushar D Chandra. 1999. Matching Events in a Content-Based Subscription System. In *Symposium on Principles of Distributed Computing (PODC)*.

[3] Jasmin Ajanovic. 2009. PCI Express 3.0 Overview. In *A Symposium on High Performance Chips (Hot Chips)*.

[4] AMD. 2019. AMD Infinity Architecture: The Foundation of the Modern Datacenter. Product Brief. amd.com/system/files/documents/LE-70001-SB-InfinityArchitecture.pdf, last accessed on 08/17/2020.

[5] AMD. 2020. AMD Crossfire™ Technology. www.amd.com/en/technologies/crossfire, last accessed on 04/14/2021.

[6] 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 of Computer Architecture (ISCA)*.

[7] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E Strom, and Daniel C Sturman. 1999. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *International Conference on Distributed Computing Systems (ICDCS)*.

[8] Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman. 1999. A Case for Message Oriented Middleware. In *International Symposium on Distributed Computing (DISC)*.

[9] Trinayan Baruah, Yifan Sun, Ali Dinçer, Md Saiful Arefin Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[10] Arkaprava Basu, Sooraj Puthoor, Shuai Che, and Bradford M Beckmann. 2016. Software Assisted Hardware Cache Coherence for Heterogeneous Processors. In *International Symposium on Memory Systems (ISMM)*.

[11] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2020. Groute: Asynchronous Multi-GPU Programming Model with Applications to Large-scale Graph Processing. *Transactions on Parallel Computing (TOPC)* 7, 3 (2020), 1–27.

[12] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. 2016. CANDY: Enabling Coherent DRAM Caches for Multi-node Systems. In *International Symposium on Microarchitecture (MICRO)*.

[13] Mohammad Dashti and Alexandra Fedorova. 2017. Analyzing Memory Management Methods on Integrated CPU-GPU Systems. In *International Symposium on Memory Management (ISMM)*.

[14] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[15] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards Expressive Publish/Subscribe Systems. In *International Conference on Extending Database Technology (EDBT)*.

[16] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. 2014. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-parallel Languages. *Transactions on Architecture and Code Optimization (TACO)* 11, 3 (2014), 1–25.

[17] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *Computing Surveys (CSUR)* 35, 2 (2003), 114–131.

[18] Françoise Fabret, H Arno Jacobsen, François Llirbat, João Pereira, Kenneth A Ross, and Dennis Shasha. 2001. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *International Conference on Management of Data (SIGMOD)*.

[19] Benedict R Gaster. 2013. HSA Memory Model. In *A Symposium on High Performance Chips (Hot Chips)*.

[20] Tom's Hardware. 2019. AMD Big Navi and RDNA 2 GPUs. tomshardware.com/news/amd-big_navi-rdna2-all-we-know, last accessed on 08/17/2020.

[21] Mark Harris. 2017. Unified Memory for CUDA Beginners. developer.nvidia.com/blog/unified-memory-cuda-beginners/, last accessed on 08/17/2020.

[22] Blake A Hechtman, Shuai Che, Derek R Hower, Yingying Tian, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. 2014. QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[23] Mark D Hill, James R Larus, Steven K Reinhardt, and David A Wood. 1992. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[24] Derek R Hower, Blake A Hechtman, Bradford M Beckmann, Benedict R Gaster, Mark D Hill, Steven K Reinhardt, and David A Wood. 2014. Heterogeneous-race-free Memory Models. In *International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[25] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[26] Nagesh B Lakshminarayana and Hyesoon Kim. 2014. Spare Register Aware Prefetching for Graph Algorithms on GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[27] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *International Symposium on Microarchitecture (MICRO)*.

[28] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *USENIX Annual Technical Conference (USENIX ATC)*.

[29] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *International Symposium on Workload Characterization (IISWC)*.

[30] Daniel Lustig and Margaret Martonosi. 2013. Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[31] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[32] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-aware GPUs. In *International Symposium on Microarchitecture (MICRO)*.

[33] Gero Mühl. 2002. *Large-Scale Content-Based Publish-Subscribe Systems*. Ph.D. Dissertation. Technische Universität Darmstadt.

[34] Harini Muthukrishnan, David Nellans, Daniel Lustig, Jeffrey Fessler, and Thomas Wenisch. 2021. Efficient Multi-GPU Shared Memory via Automatic Optimization of Fine-grained Transfers. In *International Symposium on Computer Architecture (ISCA)*.

[35] Prashant J Nair, David A Roberts, and Moinuddin K Qureshi. 2016. Citadel: Efficiently Protecting Stacked Memory from TSV and Large Granularity Failures. *Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 1–24.

[36] NVIDIA. 2013. CUDA Toolkit Documentation. docs.nvidia.com/cuda/, last accessed on 08/17/2020.

[37] NVIDIA. 2019. GP100 MMU Format. nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf, last accessed on 08/17/2020.

[38] NVIDIA. 2019. NVLink AND NVSwitch The Building Blocks of Advanced Multi-GPU Communication. nvidia.com/en-us/data-center/nvlink/, last accessed on 08/17/2020.

[39] NVIDIA. 2020. NVIDIA Ampere Architecture. www.nvidia.com/en-us/data-center/ampere-architecture/, last accessed on 04/14/2021.

[40] NVIDIA. 2020. NVIDIA DGX Systems. www.nvidia.com/en-us/data-center/dgx-systems/ last accessed on 04/14/2021.

[41] NVIDIA. 2020. NVIDIA NVLink High-Speed GPU Interconnect. nvidia.com/en-us/design-visualization/nvlink-bridges/, last accessed on 08/17/2020.

[42] NVIDIA. 2020. NVIDIA TITAN V, NVIDIA's Supercomputing GPU Architecture, Now for Your PC. www.nvidia.com/en-us/titan/titan-v/, last accessed on 08/17/2020.

[43] NVIDIA. 2020. PTX: Parallel Thread Execution ISA Version 7.0. docs.nvidia.com/cuda/pdf/ptx_isa_7.0.pdf, last accessed on 08/17/2020.

[44] Marc S Orr, Shuai Che, Ayse Yilmazer, Bradford M Beckmann, Mark D Hill, and David A Wood. 2015. Synchronization Using Remote-Scope Promotion. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[45] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. 2013. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *International Conference on Parallel Processing (ICPP)*.

[46] 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)*.

[47] Jason Power, Mark D Hill, and David A Wood. 2014. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[48] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement. In *Proceedings of the VLDB Endowment (PVLDB)*.

[49] Xiaowei Ren and Mieszko Lis. 2017. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[50] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[51] Tim C Schroeder. 2011. Peer-to-peer & Unified Virtual Addressing. In *GPU Technology Conference (GTC)*.

[52] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[53] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2015. Efficient GPU synchronization without scopes: Saying No to Complex Consistency Models. In *International Symposium on Microarchitecture (MICRO)*.

[54] 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)*.

[55] Mohammed Sourouri, Tor Gillberg, Scott B Baden, and Xing Cai. 2014. Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads. In *International Conference on Parallel and Distributed Systems (ICPADS)*.

[56] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. 2018. G-TSC: Timestamp Based Coherence for GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[57] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for Speed: Experiences Building a Trustworthy System level GPU Simulator. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[58] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *International Symposium on Microarchitecture (MICRO)*.

[59] Peng Wang. 2017. UNIFIED MEMORY ON P100. olcf.ornl.gov/wp-content/uploads/2018/02/SummitDev_Unified-Memory.pdf, last accessed on 02/14/2021.

[60] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *Principles and Practice of Parallel Programming (PPoPP)*.

[61] John Wickerson, Mark Batty, Bradford M Beckmann, and Alastair F Donaldson. 2015. Remote-scope Promotion: Clarified, Rectified, and Verified. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[62] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *International Symposium on Microarchitecture (MICRO)*.

[63] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[64] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*.