Improving Multi-GPU Strong Scaling Through Optimization of Fine-Grained Transfers

by

Harini Muthukrishnan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2022

Doctoral Committee:

Assistant Professor Ronald G. Dreslinski, Co-Chair
Thomas F. Wenisch, Co-Chair
Professor Todd Austin
Professor Jeffrey A. Fessler
Professor Scott Mahlke

Harini Muthukrishnan

harinim@umich.edu

ORCID iD: 0000-0002-5318-0856

To Dhyanalinga, for jump-starting my Self exploration.

ACKNOWLEDGEMENTS

Undoubtedly, the last five years at grad school has been the most challenging, yet the most rewarding phase of my life until now. The uncertainties of research amplified by the COVID pandemic, the pain of rejection and disappointment, the fulfillment of taking an idea to its completion, the relief of success, and learning to keep my pride aside to seek help when required have all made the journey worthwhile. I would like to thank everyone, a few mentioned below, but many unmentioned, who have helped me evolve into a better researcher and a much more balanced human-being over the past few years.

First and foremost, I would like to thank my advisor, Prof. Thomas Wenisch for offering me a place in his lab, for teaching me the basics of research, and for continuing to actively contribute to my research despite his career change in the later part of my Ph.D. My industry collaborators, David Nellans and Daniel Lustig mentored and supported me in many ways: from being active contributors to my research to helping me improve my written and oral communication skills. Thanks to my dissertation committee for playing a supportive role in this journey with different members contributing at different stages. The work I did in the medical imaging domain with Prof. Jeff Fessler helped me identify the direction of my Ph.D. The inputs from Prof. Ron Dreslinski, Prof. Todd Austin, Prof. Scott Mahlke and Prof. Tim Rogers helped me better structure the last part of my thesis.

Thanks to my parents for always prioritizing my academics and for trusting and believing in me my whole life. The amount of importance they gave to my education was instrumental in helping me realize its value. Thanks to my kid sister for valuing who I am and being very supportive.

The longing to explore my inner nature has been a huge part of my life ever since I spent a few minutes in the Dhyanalinga temple twelve years ago. My gratitude to numerous teachers who have since then assisted my seeking.

My journey was enriched by the entourage of friends I made over the past few years. My gratitude to Sriram, Krishna and Sangeeta for all the amazing conversations and to Sivakumar Ardhanari for teaching me the basics of Computer Architecture and being a great mentor. I would also like to thank the citizens of Wenisch lab a.k.a. the *Sanctuary lab* for being awesome labmates.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Recent GPU architectural enhancements enable easy multi-GPU programming by exposing the GPUs' aggregate memory as a single shared address space, while allowing direct remote data accesses among them. However, despite dramatic improvements in inter-GPU latency and bandwidth, inter-GPU communication remains the largest architectural bottleneck in multi-GPU systems. With hundreds of thousands of independent concurrently executing threads, maximizing interconnect utilization without degrading computational efficiency when strong-scaling HPC workloads is an open problem.

In this dissertation, I explore proactive, peer-to-peer writes as the communication paradigm for improved multi-GPU strong scaling. Proactive writes enable a natural overlap of compute and communication by performing remote transfers as and when the data is generated. This results in improved utilization of compute and communication bandwidth over the traditional mechanisms of data movement such as bulk transfers and on-demand loads. However, existing GPU and interconnect architectures are not designed to suit such transfers, resulting in significant bandwidth wastage and programming complexity. To overcome these limitations, I propose a transfer mechanism, a memory management technique and an interconnect architecture to improve the programmability and performance of proactive peer-to-peer writes. These solutions based on hardware/software co-design offer huge performance advantages over the existing multi-GPU architecture while providing a simple and intuitive programming interface.

In the first part of the dissertation, I propose PROACT, an inter-GPU communication mechanism which enables remote memory transfers with the programmability and pipeline advantages of peer-to-peer stores, while achieving interconnect efficiency that rivals bulk

DMA transfers. I describe both hardware and software implementations of PROACT and demonstrate the effectiveness of a PROACT software prototype on three generations of GPU hardware and interconnects. I also present a real-life case study of PROACT, demonstrating its benefits to improve strong scaling for Model-Based Iterative Reconstruction (MBIR), a computationally expensive algorithm used in medical imaging.

In the next part of the dissertation, I propose a novel memory management solution GPS (GPU Publish Subscribe), to efficiently orchestrate inter-GPU communication using proactive data transfers. GPS enables the programmability of multi-GPU shared memory with the performance of GPU-local memory by automatically tracking the data accesses performed by each GPU, maintaining duplicate physical replicas of shared regions in local memory, and publishing updates to the replicas among GPUs. GPS lives within the existing GPU memory consistency model but takes full advantage of its relaxed nature in order to deliver performance improvements.

In the last part of the dissertation, I discuss my work to design and evaluate FINEPACK, a set of I/O interconnect and GPU hardware enhancements for high fine-grained access efficiency. Existing and emerging multi-GPU interconnects focus primarily on bulk transfers or coherence and exhibit poor performance for small accesses due to high protocol overheads. FINEPACK provides improved efficiency by packing fine-grained transfers into a giant packet, enabling sharing of common protocol headers and thus reducing the wastage of precious interconnect bandwidth. I evaluate FINEPACK on a system comprising 4 Volta GPUs on a PCIe 4.0 interconnect to show it improves interconnect efficiency for small peer-to-peer stores by 5×, results in 4-GPU strong scaling performance 1.5× better than traditional DMA based multi-GPU programming, and comes within 77% of maximum achievable strong scaling performance.

CHAPTER I

Introduction

Graphics Processing Units (GPUs) are becoming ubiquitous for accelerating high-performance computing applications, since they enable enormous computational density. Yet, many applications exhibit more parallelism than what a single GPU can provide; these applications can spawn more thread blocks than there are compute cores on current GPUs and generate more memory parallelism than the available local memory bandwidth. One way to accelerate such applications further is via strong scaling—-to spread the excess computation to multiple GPUs.

A key challenge in multi-GPU parallelization lies in managing the limited bandwidth of the inter-GPU interconnect. Each GPU provides enormous bandwidth to locally-attached GPU memory; far greater than the interconnect bandwidth available between GPUs. The NUMA bandwidth and latency disparity between local and remote memory is most pronounced in systems where GPUs are connected via low-cost PCIe, but these systems are typically limited to just 2–4 GPUs. In high-end systems with as many as 16 GPUs [112], improved interconnects provide much higher bandwidth [114, 115] but there are also $4\times$ as many remote GPUs that must coordinate efficiently. Figure 1.1 demonstrates the variation in local, aggregate remote, and peer-to-peer bandwidths across several generations of GPUs. Across these systems, the ratio of local to remote memory bandwidth can be as high as 10:1 on Kepler based PCIe systems and is still 3:1 on NVIDIA's recent 16-GPU DGX-2 system.

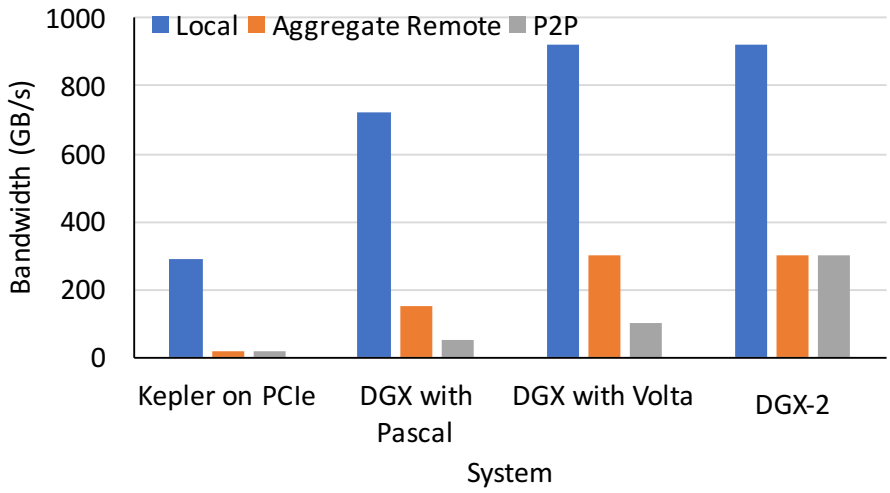


Figure 1.1: Variation in local and remote bandwidths across GPU generations.

The disparity between local and remote memory bandwidth makes data locality management a significant challenge in strong scaling multi-GPU applications. This challenge is illustrated by Figure 1.2, which shows the average speedup over a single GPU on an NVIDIA DGX-Station system with four Volta GPUs [113], for five applications from a variety of domains (described later in Section 2.4). In orange, we show potential speedup when we consider computation scaling alone and neglect the cost of data transfer (all accesses are GPU-local). In blue, we show the actual performance when the applications are naively partitioned across GPUs, using direct loads and stores to access remote GPU memory. Whereas computational performance scales close to the linear ideal, the overhead of remote loads and stores is so high that the applications suffer a net slowdown. Efficiently managing inter-GPU bandwidth—ensuring the loads during computational kernels access local GPU memory—is critical to enable multi-GPU scaling.

One way to improve locality is to perform judicious and explicit memory management among GPUs. As GPU architectures and the CUDA programming interface have matured, an increasingly diverse set of mechanisms, such as direct memory access (DMA) engines [145], peer-to-peer PCIe transfers [138], multiple CUDA streams [131], unified memory (UM) [110], and dynamic parallelism [62] have been added to the GPU ecosystem

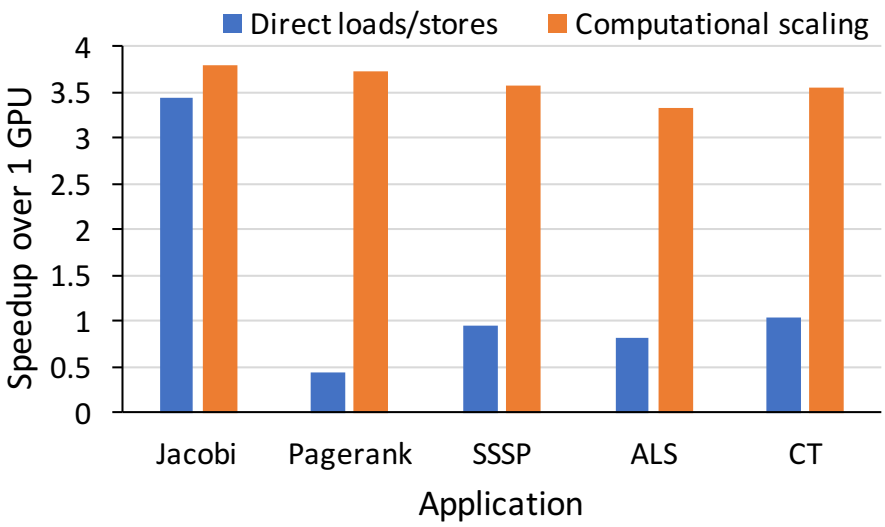


Figure 1.2: Actual speedup with direct loads/stores vs. hypothetical speedup of computation alone on a NVIDIA DGX-Station with 4 GV100 Volta GPUs.

to help enable the programmer to maximize performance. These mechanisms lead to an astonishing diversity of approaches to move data among GPUs.

The simplest (from a programmer's perspective) approach for coordinating data transfers among GPUs is to rely upon GPUs' automatic memory management runtime, called Unified Memory; UM automatically moves data among the CPU and GPUs, on demand, as it is accessed. While effective at improving programmability, we will show that the overheads of page-fault–based memory management use the limited interconnect resource poorly and are inappropriate for high performance multi-GPU systems; they typically lead to degradation relative to a single GPU. This observation motivates us to study and optimize the space of programmer-directed memory management in multi-GPU systems.

Existing GPU interconnect architectures are optimized to explicitly transfer data between GPUs is via host-initiated peer-to-peer `cudaMemcpy()` invocation between kernels executing on the GPUs. This technique is simple to implement, uses efficient DMA hardware, and typically occurs between kernel invocations. By occurring only between kernel executions, there is an implicit GPU-wide barrier before data transfer, easing the programmer burden for algorithm design and data movement coordination. However, as GPU workloads scale

beyond a small number of GPUs, interleaving data movement between kernel invocations results in frequent multi-gigabyte all-to-all copies and poor utilization of the limited interconnect bandwidth during kernel execution; these bulk transfers rapidly degrade multi-GPU scalability due to Amdahl's Law.

To enable performance scalability on many-GPU systems, we argue that both architects and GPU programmers must change their mindset about how, and when, GPU execution and data transfers should occur. Application designers must embrace solutions that overlap data transfers with GPU execution by streaming data among GPUs as it becomes ready in a fine grained producer–consumer relationship. In this dissertation, I will show that this simple idea entails several significant challenges since the existing communication mechanisms, memory management runtime and interconnect architectures are not suited to perform for such transfers. To overcome these limitations, I partner with industry experts to develop novel architectural and programming model modifications through hardware/software co-design to improve strong scaling performance. Through simulations and software prototyping of multiple real world applications, I achieve orders of magnitude improvements in both performance and interconnect efficiency versus systems available today.

1.1 Contributions of the dissertation

Throughout the remainder of this dissertation, I detail my major contributions spanning across the hardware and software stack to tackle the challenges in adopting proactive fine-grained transfers as a communication paradigm to improve multi-GPU strong scaling. First, I will detail PROACT [97], a joint compile and runtime system that transparently fine-tunes inter-GPU data movement for each application's needs, thus achieving the interconnect efficiency of bulk transfers at the programming simplicity of peer-to-peer stores. Next, I will demonstrate how GPS [95], a HW/SW memory management technique, employs selective page replication and proactive remote stores to improve read locality while conserving the interconnect bandwidth. Finally, I will discuss FinePack [96], a set of architectural

enhancements to overcome the limitations of existing multi-GPU interconnects to perform small transfers efficiently.

CHAPTER II

PROACT: Efficient Multi-GPU Shared Memory via Automatic Optimization of Fine-grained Transfers*

2.1 Introduction

Despite advancements in GPU architecture and programming models, achieving peak performance on multi-GPU systems remains a challenge for GPU programmers [21, 39, 148, 102, 130, 88]. To efficiently parallelize applications across a multi-GPU system, developers typically distribute an application's data structures across each GPU's physical memory. Programs are also designed to operate in logical phases, alternating between periods of heavy computation accessing mostly locally-available data and periods of data distribution and synchronization among the GPUs. Unfortunately, this typically results in inefficient use of resources: interconnects sit idle during computation phases and compute units sit idle during communication phases.

2.1.1 Challenges of existing paradigms

To obtain high interconnect utilization and maximize performance on multi-GPU systems, developers are forced to dedicate a substantial manual effort to performance-tuning and re-architecting their applications to ensure optimal data distribution, inter-GPU com-

*This chapter based on [97]

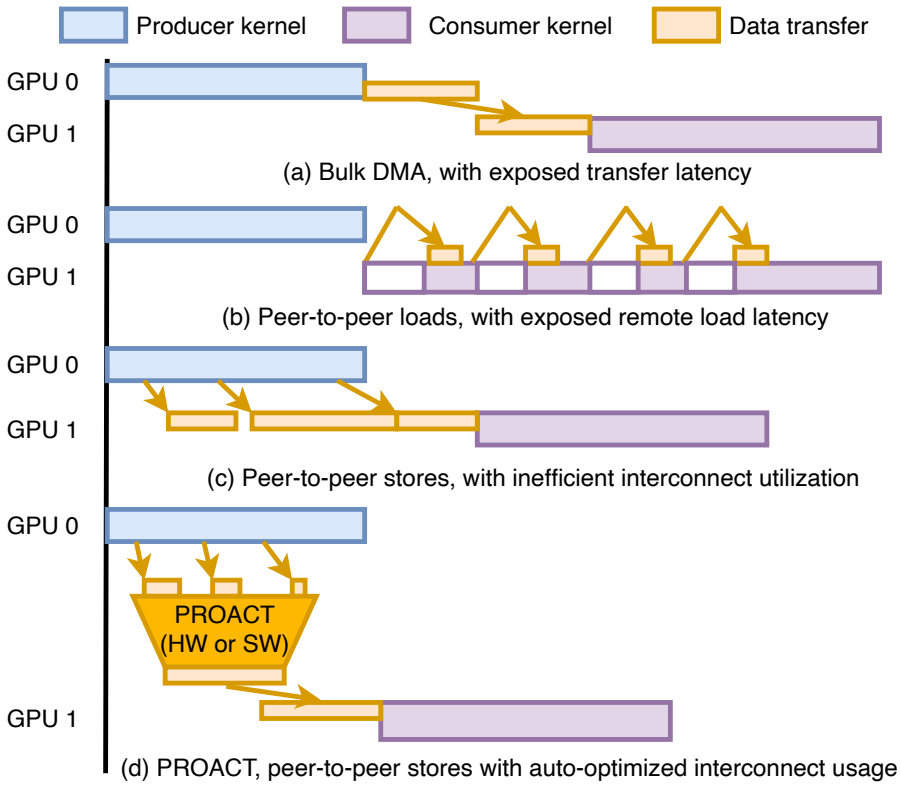


Figure 2.1: Different multi-GPU communication paradigms

munication, and synchronization management for correctness. These steps often require detailed knowledge of internal GPU architecture, GPU interconnects, and the memory access patterns of the application. As a result, peak multi-GPU performance is often only achievable by "ninja" programmers.

One major impediment to improving multi-GPU performance is the difficulty in efficiently overlapping GPU computation with communication phases, leading to system resource under-utilization. Historically multi-GPU applications have used bulk DMA-based communication among GPUs because these DMA transfers provide high interconnect utilization during the transfer phases. Unfortunately, due to bulk synchronization, GPU computation cannot generally take place during these communication phases. Figure 2.1(a) depicts a typical DMA-based transfer between 2 GPUs. The DMA transfer occurs between the producer and consumer kernels on the GPUs, exposing the data transfer latency.

More recently, fine-grained peer-to-peer (P2P) transfers enable GPU compute units

to issue loads and stores directly to the physical memory of remote GPUs. These peer to peer (P2P) transfers enable inherent overlap of compute and communication, but P2P transfers are inefficient on current GPU interconnects leading to gross under-utilization of the inter-GPU interconnect. Depicted in Figure 2.1(b), P2P loads often result in the consumer kernel stalling while waiting for remote loads to complete, due to interconnect latency, negatively impacting performance. Shown in Figure 2.1(c), P2P stores between GPUs often are generated at sub-cacheline granularity and with sporadic access patterns. Although some store latency is hidden, due to protocol packetization overheads, these stores do not use the GPU-interconnect efficiently.

2.1.2 Contributions

This work propose PROACT, a joint compile-time and runtime system that combines the flexibility of peer-to-peer stores with the interconnect efficiency of bulk transfers, as shown in Figure 2.1(d). PROACT exposes an easy to use P2P-store programming model to developers, but leverages profiling and GPU-runtime support to track data generation, perform transfer coalescing, and dynamically issue transfers over the interconnect to achieve high utilization. PROACT provides efficient overlap of per-GPU computation and inter-GPU data transfer by performing the transfers proactively soon after data generation and can be supported with either dedicated hardware or software (at the expense of some GPU computation resources).

To evaluate the PROACT approach of balancing fine grained transfers with intelligent interconnect utilization, we provide a comprehensive performance comparison to other common multi-GPU programming paradigms that use automatic GPU memory management, barrier-based GPU transfers, and fine-grained data accesses. To demonstrate the performance of PROACT we implement a software prototype of our design on real hardware and show that even while consuming GPU resources (an overhead which would be eliminated with future HW support), PROACT is the best performing approach to multi-GPU programming,

enabling strong scaling to an 11× mean performance improvement on a prototype 16-GPU system—5.3× better than next-best alternative, a standard bulk-synchronous approach.

2.2 Background

2.2.1 CUDA programming model

In the CUDA programming model [101], a *kernel* comprises many thousands of threads grouped into *blocks*. Sets of threads (32 in recent GPUs) within a block are grouped together into *warps*, which are scheduled by hardware and execute in lock-step on a Streaming Multiprocessor (SM).

A central tenet of the CUDA programming model—critical to enabling switch-on-miss multithreading—is that the warps in a block may execute in any order. As a consequence, developers typically ensure that all GPU threads within a kernel are independent. Independence creates an easy opportunity to spread execution across multiple GPUs if data structures in local GPU memory can be replicated or partitioned, scaling both the compute and memory bandwidth available to the application. However, splitting an application across GPUs creates a new bottleneck—exchanging data over the shared interconnect.

2.2.2 Inter-GPU Communication Mechanisms

Regardless of the amount of remote data a multi-GPU program may access during its execution, GPU programmers must choose between several mechanisms to perform these accesses or copies, and each mechanism has unique performance characteristics. We describe these mechanisms and their performance implications using NVIDIA's terminology to provide specific examples, though the concepts themselves are general. Pedagogically, we refer to a kernel that requires data produced by another GPU as a consumer kernel and the kernel that generates data for kernels on other GPU(s) as a producer kernel. In practice, a single kernel is often both the producer of some data and a consumer of other

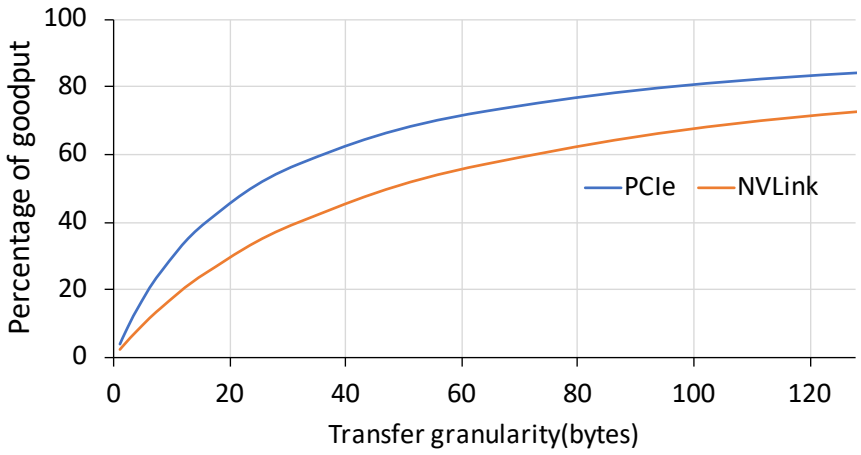


Figure 2.2: Interconnect goodput vs. write transfer granularity.

data simultaneously.

DMA-based bulk transfers: When using bulk DMA-based transfers between GPUs, data transfer is typically explicitly invoked from the host program executing on the CPU via `cudaMemcpy()`. The copy is scheduled following the completion of the producer kernel, before invocation of the consumer kernel. The `cudaMemcpy()` call invokes the GPU's hardware DMA engine to transfer data directly between the GPUs' memories without the need for the transfer to reflect off the CPU's memory system.

This paradigm can saturate most GPU interconnects when transferring very large granularity data and ensures that subsequent accesses made by the consumer kernel will be serviced at high bandwidth from within the consumer GPU's local physical memory. However, this method is not suitable for small (several cache lines) or medium (several KB) sized transfers due to the high initialization and synchronization overhead of returning to the host program and then programming the DMA engine. Each of these steps can consume several microseconds [45], which dominates the data transfer time itself. Though programmers can attempt to interleave computation and communication using DMA-based transfers, it requires substantial programmer expertise and effort.

Peer-to-peer (P2P) GPU accesses: In modern multi-GPU systems, individual GPUs are capable of directly reading and writing the physical memory of peer GPUs without GPU run-

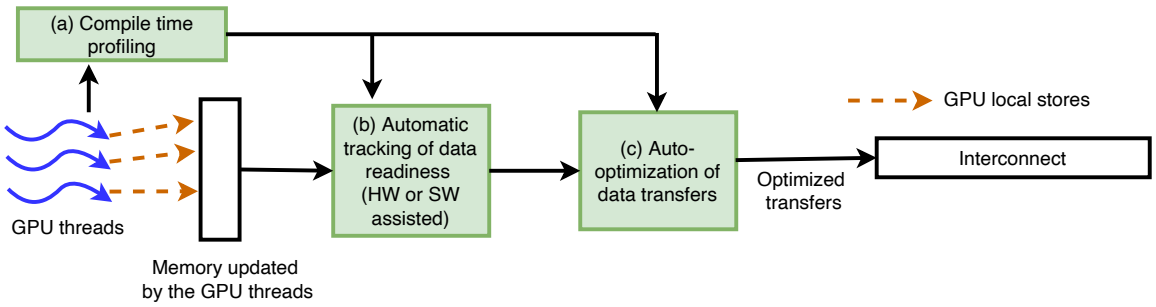


Figure 2.3: Overview of the major logical PROACT components.

time intervention or host CPU synchronization. The advantages of performing direct remote accesses using GPU threads is that there is no initiation overhead (such as programming a DMA engine) and the accesses can be performed while unrelated computation continues in parallel, within a given thread and without stalling other threads. However, when loads are performed from remote memory, these P2P loads often stall thread execution beyond the GPUs' ability to hide the load latency with multi-threading due to the longer latency of the inter-GPU interconnect as compared to local memory. As these load stalls build up within the GPU's memory system, they eventually consume precious GPU resources that would otherwise have been used to make execution progress.

Unlike P2P reads that eventually may stall each issuing thread in the GPU, P2P writes almost never stall the issuing thread because there is no implicit dependence within program execution. As such, they consume substantially fewer on-chip resources. We believe the non-blocking nature of P2P stores makes them fundamentally the most efficient way to perform multi-GPU communication. P2P stores are not a panacea, however. Straight-forward use of P2P stores will typically result in numerous small writes (as small as one byte) issued over the inter-GPU interconnect, which results in poor interconnect utilization. PROACT adopts P2P stores as the basis of its programming model and the major contribution of this work is designing a system that can overcome poor interconnect efficiency when performing small P2P writes.

Programmatic communication libraries: In an effort to ease development burden it is common for GPU programmers to use use GPU-aware libraries such as MPI [158],

NVIDIA's NCCL [58], or NVIDIA'a NVSHMEM [124] to provide universal inter-GPU communication and synchronization patterns, irrespective of the interconnect technology and topologies on which the code is run. In turn, under the hood these libraries may selectively use bulk DMA transfers or P2P accesses, combined with library-controlled memory management to optimize communication between GPUs.

Although these libraries continue to be optimized, there is little they can do to avoid DMA initiation overhead when performing large transfers. To the best of our knowledge none of these libraries attempt to aggregate fine grained transfers intelligently to improve interconnect efficiency. We will show that brief compile-time profiling of a system and runtime optimization for interconnect utilization can provide substantial performance improvement over the underlying mechanisms used by these libraries, and without loss of generality, the PROACT technique could be implemented as a new back end to many of these commonly used libraries. We discuss several additional GPU communication libraries in Section 2.6.

2.2.3 Inter-GPU Interconnect Efficiency

System architects and builders have a range of interconnect technologies to choose from while designing multi-GPU systems. For over a decade and still today, PCIe [140] has been the dominant interconnect used to attach peripherals and accelerators to CPUs, and to each other. More recently, NVLink[114] was designed by NVIDIA as a dedicated GPU interconnect that targeted higher bandwidth and improved scalability over PCIe. Other high performance interconnects such as Infiniband [158] and AMD's Infinity Fabric [162] exist but target large scale networking and CPU-CPU connections respectively. We choose to evaluate PCIe as the most common GPU interconnect used today and NVIDIA's NVLink as the highest performing dedicated GPU interconnect.

Despite both evolutionary (PCIe) and clean slate (NVLink) design progression where both protocols support direct peer to peer accesses between GPUs, both PCIe and NVLink

have poor efficiency when performing small accesses. Figure 2.2 shows the percentage of goodput achieved on the interconnect, i.e., the percentage of useful data delivered over PCIe and NVLink interconnects for varying store granularities. Both interconnect technologies provide high efficiency for transfers with greater than 128 bytes (a common cache line size) but drop off dramatically at smaller transfer sizes. Interconnect efficiency decreases at these smaller granularities because protocol packetization overheads dominate the effective goodput and the transfer efficiency falls as low as 8% on NVLink and 14% on PCIe for 4-byte stores. Therefore, while P2P writes may fundamentally be the most efficient way to transfer data from a latency-hiding perspective, to improve overall multi-GPU performance, dramatic improvements in interconnect efficiency must be achieved.

2.3 PROACT Design and Implementation

PROACT attempts to bridge the benefits of DMA-based bulk copies and peer-to-peer accesses in multi-GPU systems to provide the interconnect efficiency of large transfers with the non-blocking semantics of SM initiated peer-to-peer stores. PROACT improves the performance of multi-GPU systems by (1) balancing the overlap of data transfers with GPU computation, (2) maximizing the opportunity for write coalescing, (3) smoothing interconnect utilization over time to ensure no bandwidth is wasted, and (4) increasing interconnect efficiency by ensuring communication occurs at sufficiently large granularities.

To provide these benefits for a range of applications with varying data access patterns, PROACT needs to make the following design choices depending on the application and system architecture.

- **Transfer mechanism**: GPUs provide different ways to transfer data among memories. The best approach overlaps compute with communication while providing high interconnect utilization and efficiency.

- **Transfer granularity**: Data transfer mechanisms vary in efficiency as a function of

granularity. Performing transfers in coarser chunks can reduce initiation overhead, but may result in a large *tail chunk* that extends past the end of computation, delaying the next phase. On the other hand, transferring data as fine-grained chunks can lead to poor interconnect efficiency as described in Section 2.2.3.

- **Transfer resources**: Performing data transfers by employing all threads in the system typically leads to interconnect inefficiencies and compute stalls, but developing applications that use per-thread warp specialization so that only a subset of the threads are writing data to remote GPUs is both difficult to implement and error prone. Thus, we need to automatically identify the appropriate amount of GPU resources to perform the transfers and hide this complexity from programmers as much as possible.

- **Tracking data generation**: CUDA does not guarantee thread ordering. Hence, we need to develop an appropriate mechanism to track data production and then trigger efficient transfers.

PROACT has three primary components that coordinate up and down the software and hardware stack to make these design choices and maximize throughput. Figure 2.3 provides an overview of these building blocks. They are (a) a compile-time profiler to determine the different PROACT parameters required to achieve optimized transfer efficiency (b) a tracking unit to monitor data readiness (c) a data transfer mechanism that initiates optimized transfers. Each component is described in further detail below.

2.3.1 Optimizing Transfer Efficiency via Profiling

The first step in improving interconnect efficiency is identifying how to extract performance from the multi-GPU interconnect without hampering GPU compute throughput. If the granularity of P2P stores is too small, they offer poor interconnect efficiency, as described earlier in Section 2.2.3. If a sufficient number of writes are not in flight on the interconnect at any given time, bandwidth will go underutilized.

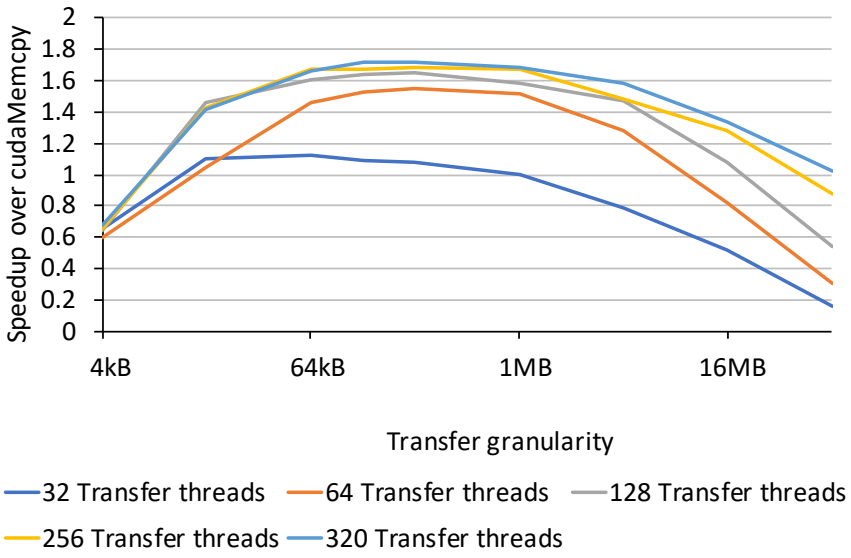


Figure 2.4: Automatic profiling showing workload throughput vs GPU transfer threads and aggregate transfer size.

Perhaps unsurprisingly, performance is a complicated function of multiple parameters. Figure 2.4 provides an example of the sensitivity to the number of transfer threads (see Section 2.3.3) and aggregate transfer granularity on an NVIDIA Kepler based system. In this case the best application throughput is obtained for transfer granularities between 64kB and 1MB when the number of threads used for the transfers is more than 128. Using additional threads (or in fact all threads on the system) to saturate the interconnect results in decreased compute efficiency, without any improvement in interconnect utilization. Each GPU generation and interconnect type has a distinct number of GPU threads that are needed to saturate the interconnect bandwidth. By examining these profiles, PROACT can choose the number of transfer threads for a given system to maximize interconnect bandwidth while minimizing how many GPU execution lanes are used for data transfer.

To facilitate exploration of the complex design space, as shown in Figure 2.3(a), PROACT contains a compile time software profiling suite that identifies the appropriate balance of the many competing factors that affect performance on a given GPU and interconnect. The PROACT profiler performs a parameter sweep and analyzes application

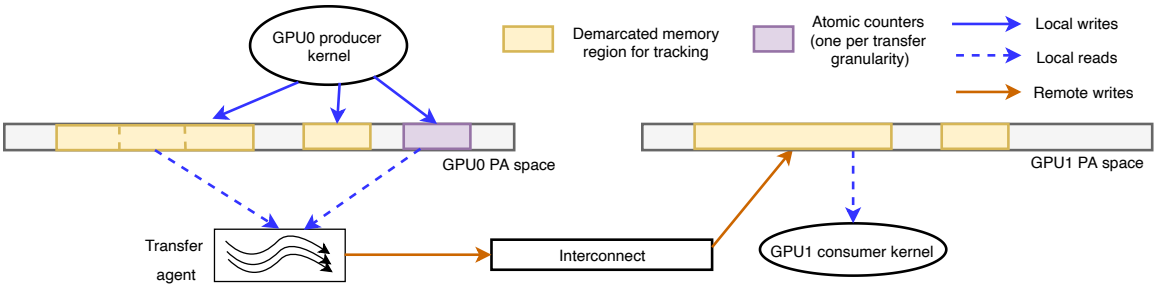


Figure 2.5: Decoupled local data generation with region tracking for decoupled transfers.

performance across PROACT's different transfer mechanisms (described later in Section 2.3.3), and varying the transfer chunk size and transfer thread count, to identify the best configuration for a given application and system. It then configures the application compilation to emit code that is most efficient according to the profiling results.

2.3.2 Tracking Local Data Transfer Readiness

PROACT presents users with a programming model that encourages the use of P2P stores, however transmitting data over the interconnect at fine granularity leads to poor interconnect efficiency. We propose that remote GPU transfers should be decoupled from each individual thread's writes by aggregating these writes into local GPU memory before being transferred to remote GPUs. Shown in Figure 2.5, PROACT provides a staging mechanism whereby GPU threads individually write to a local buffer (as if it was remote). This locally written data is then propagated to remote GPUs by an asynchronously operating transfer agent. By decoupling the transfer of data over the interconnect from the the generation of data locally, PROACT can optimize the transfer timing, granularity, and mechanisms it chooses to employ (based on profiling).

The NVIDIA GPU memory model only requires weak writes and strong writes with `cta` and `gpu` scopes to be visible at remote GPUs no later than the succeeding global synchronization barrier. PROACT exploits this slack by aggregating data in local memory until all writes to a transfer chunk complete, only then pushing out data to keep the interconnect saturated with high efficiency transfers. Strong writes with `sys`-scope are typically used as

global memory synchronization instructions rather than for performing data transfers and hence do not fall under the purview of PROACT.

The existence of a decoupled transfer mechanism implies a need to track when a transfer is ready to begin. Whereas direct remote GPU stores require no special data tracking mechanism (i.e., remote stores are simply issued by the producer kernel), decoupling remote transfers from their local generation imposes a need for explicit mechanisms to track data generation and trigger transfers. By employing structured programming, the basic mechanism that PROACT uses to track data destined for other GPUs is relatively straightforward: a set of atomically-updated per-chunk counters track when stores are generated by the producer thread as shown in Figure 2.5. While ultra-fast hardware support for tracking the readiness of PROACT's local memory buffers would be beneficial, it is not necessary because GPUs today already support atomic counters in memory (typically implemented in the GPU's L2 for performance reasons). Thus, while less efficient than custom hardware support, it is possible to prototype the data tracking portion of PROACT entirely in software on today's GPUs.

Because GPU programmers are familiar with a producer-consumer paradigm, it is rare that data structures are globally shared among GPUs with no coordination regarding locality. This makes porting applications to the PROACT program structure easy. For legacy multi-GPU programs that use explicit writes to local buffers, followed by manual or library-managed copies to a remote GPU, the computation phase of the application will remain unchanged, while the communication phase is simply elided because PROACT now orchestrates the transfers automatically during kernel execution.

Figure 2.5 shows a simplified example in which an application's memory is laid out linearly. As the producer kernel produces data it also updates the atomic counters, maintained at a different location in memory. The transfer agent polls theses counters and when the expected value is observed, the transfer is started. Linear arrangements of data in memory are not required but contiguous buffers result in better performance and reduce the amount

of storage overhead needed for the atomic counters versus the buffers themselves. We note that, because PROACT typically chooses transfer granularities of 64kB–1MB (shown later in Table 2.2) the storage overhead of the counters themselves is not a significant concern.

2.3.3 Choosing the Decoupled Data Transfer Mechanism

By decoupling data transfer from local data generation, we have the flexibility to chose an inter-GPU transfer mechanism and transfer granularity that can maximize bandwidth utilization between GPUs. Because of the initiation overhead of the GPU DMA engine it not appropriate for frequent data movement among GPUs, leaving us with three alternatives.

Long-lived Software Threads: In this technique, a small configurable number of GPU warps are specialized to perform data transfers at a given granularity. Auto-generated and managed entirely by the PROACT runtime, this independently launched long-lived kernel polls the structured data-ready atomic counters that are updated by the application's producer kernels. When ready, the transfer threads read a chunk from the local staging array and perform remote writes to the target buffer on the target GPU. While PROACT's profiler tries to minimize the number of transfer threads required to maximize interconnect performance, ultimately these threads and the polling of the counters (in a SW implementation) compete with the compute kernel(s) for execution throughput. Memory consistency between the producer threads and the decoupled transfer threads is enforced by using existing GPU memory ordering fences in our PROACT SW-prototype.

Dynamic Kernel Launches: DMA offload is high latency and consumes no GPU compute resources, while long-lived software threads dedicated to transfers are low latency but consume GPU compute bandwidth via polling even when no data is being transferred. CUDA Dynamic Parallelism (CDP) [63] presents another transfer agent mechanism, by allowing a parent kernel to launch child kernels as needed. In contrast to using long-lived threads, which consume compute bandwidth in the polling loops that monitor for chunk transfer readiness, dynamic kernels are launched only when a chunk is ready for transfer and

consume compute resources only during the transfer itself. The CUDA runtime guarantees that parent and child kernels have a fully consistent view of global memory when the child kernel starts and ends so no additional synchronization is needed. PROACT implements CDP-based transfers an alternative to long-lived threads to avoid the compute bandwidth cost of polling the transfer bitmaps. The trade-off, however, is that CDP kernel launch has a higher initiation latency than the long-lived transfer kernels, though still lower than DMA based transfers.

To use CDP based transfers, when the atomic data counter indicates that a data chunk is ready for transfer, PROACT's instrumentation in the producer kernel launches a dynamic kernel (with a pre-configured number of threads) to perform the transfer. This dynamic kernel transfers the data chunks to all destination GPUs using tightly packed SM store instructions.

Direct Inline Stores: To evaluate the full design space of options, our PROACT software prototype also includes an *inline* version wherein native P2P stores issue remote writes to distribute data to remote GPUs as data are produced (i.e., without deferring to decoupled transfer threads), though we do not expect this to perform well in practice. Direct inline stores have the advantage that they spread remote writes over the course of the producer kernel without the tracking overhead of decoupled transfers, smoothing interconnect utilization as long as remote writes are evenly distributed within the kernel execution. Because GPU stores are usually non-blocking and can coalesce adjacent writes, remote transfer latency is hidden unless queuing resources are exhausted. Functionally a memory ordering barrier at the end of the producer kernel is needed ensures all transfers are complete and subsequent accesses by the consumer kernel can then be made locally. If data generated by the producer has poor spatial locality, write coalescing may fail, inflating the interconnect bandwidth required to transfer the applications data, relative to the decoupled transfer mechanisms.

2.3.4 Hardware Support for PROACT

In this study, we implement a SW prototype of PROACT because it allows us to validate the concept across wide range of GPU generations and interconnects, which would be impossible to simulate in reasonable time. With additional hardware support, we envision an implementation where the data readiness counters are provisioned in a dedicated memory structure that are initialized by the PROACT runtime and associated with the base and bound of local transfer buffers located in GPU memory. Local writes issued to a transfer buffer's address range automatically update the corresponding counter, replacing the explicit instructions added by PROACT instrumentation to maintain counters in our prototype. When a counter decrements to zero, hardware signals a transfer agent to initiate a transfer of the corresponding buffer. The transfer agent itself can be realized as a simplified DMA engine with descriptors for the geometry of each transfer prepared in advance in memory by the PROACT runtime. The salient aspect of a hardware design is that transfers are triggered automatically and without the need for interaction with GPU drivers running on the host CPU. Because PROACT can be prototyped entirely in software (with SW overheads quantified later in Figure 2.8), we leave microarchictural details of a PROACT hardware realization for future work and focus on demonstrating the efficacy and generality of the PROACT approach across numerous GPU and interconnect topologies using our software prototype.

2.4 Experimental Methodology

To evaluate PROACT's software prototype we perform our experiments across three 4-GPU platforms (4x Kepler, 4x Pascal, and 4x Volta) with key characteristics described in Table 2.1 and one 16-GPU platform (16x Volta) [141, 105, 36, 106, 107, 108].

System	4x Kepler	4x Pascal	4x Volta	16x Volta
GPU	Tesla K40m	Tesla P100	Tesla V100	Tesla V100
GPU Arch	Kepler	Pascal	Volta	Volta
#GPUs	4	4	4	16
Interconnect	PCIe3.0	NVLink	NVLink2	NVSwitch
Bidirectional BW per GPU	16GB/s aggregate	150GB/s aggregate	300GB/s aggregate	300GB/s aggregate
#Cores (SMs)	15	56	80	80
TFLOPS	1.43	5.3	7.8	7.8
BW (GB/sec)	288.4	720	920	920
Mem Cap (GB)	12	16	16	32

Table 2.1: Key characteristics of the GPUs used in the experiments and their interconnect topologies in test systems.

2.4.1 PROACT Code Framework

To aid users, the PROACT prototype provides a skeleton code framework, or template, that can be adapted for use by each target application. The skeleton framework comprises two components: an initialization interface that runs on the host and prepares for proactive transfers and a wrapper interface that provides runtime support for progress tracking and data transfer initiation.

The initialization interface is invoked from the host prior to kernel launch. It allocates the atomic counters to track data generation and also manages a mapping function that determines how the data to be distributed between GPUs is laid out in the virtual address space. This mapping function associates the writes performed by a thread with a particular atomic counter that tracks progress for a particular transfer granule.

The wrapper interface is shown in Algorithm 1. The wrapper encloses the user's compute kernel with code to track progress. It first invokes the user's kernel to produce a particular datum or data range (line 2). Next, a barrier at the end of the block ensures all warps in the block complete (line 3). Following this barrier, the first thread in each block decrements a corresponding progress-tracking counter. When this counter reaches zero, the wrapper invokes either a copy kernel on a separate stream or sets a bit in the ready bitmap or hardware transfer unit to indicate a data chunk is ready.

Algorithm 1 PROACT Wrapper

```
1:  procedure PROACT WRAPPER(kernel_parameters[])
2:      compute_kernel(kernel_parameters[])
3:      __syncthreads()
4:      if is_first_thread_in_block() then
5:          chunk = block_to_chunk_mapping(blockId)
6:          atomicDec cntrs[chunk]
7:          if cntrs[chunk] == 0 then
8:              launch COPY_INITIATOR()
9:  procedure COPY_INITIATOR
10:     if DedicatedHW then
11:         atomicDec atomic_unit[chunk]
12:     else if PersistentKernels then
13:         set bitmap[chunk]
14:     else if DynamicKernelLaunch then
15:         launch dynamic_copy_kernel()
```

2.4.2 Evaluated Design Alternatives

To demonstrate the dynamic benefits of PROACT, we compare it to a number of different static multi-GPU programming approaches.

cudaMemcpy: In this paradigm, the computation kernel is followed by a `cudaMemcpy` call that duplicates data structures among all GPUs as needed. By the initiation of the following kernel, all data structures accessed by that kernel are resident in local GPU memory; there are no remote accesses. However, there is also no overlap between data transfers and compute. This approach makes great utilization of the interconnect when the transfer is in progress, but SMs remain idle during that time.

Unified Memory (UM): Unified memory was introduced to aid in the rapid development of GPU applications. We port our workload implementations to use unified memory by replacing conventional memory allocations with unified memory allocations, removing explicit data transfers, and adding the required inter-GPU synchronization. UM provides hint APIs to allow expert users to try to avoid the page faults and their large respective overheads. We hand-tested a variety of hinting strategies, including data pre-fetching, read-replication, and pre-population of the GPU page tables to reduce fault overheads, making

a best effort attempt to optimize for each application. We note that though several of the workloads benefit substantially from read-replication hints, UM still does not make efficient use of the interconnect, especially for applications with sporadic access patterns.

PROACT-inline: PROACT-inline is a pattern within the PROACT framework that performs remote writes as part of the data generation kernel, rather than performing transfers on decoupled transfer threads. Remote stores are injected directly into the source kernel to push data to remote GPUs, rather than relying on the decoupled transfer agent to optimize the transfers.

PROACT-decoupled: PROACT-decoupled uses the full flexibility of the PROACT mechanisms, using the profiling tools to select the best data transfer method and parameters for each platform and application.

Infinite Interconnect BW: Finally, we include a limit study that shows performance when data transfers are instantaneous. This represents the maximum speedup that applications can attain from optimizing data movement. Here applications enjoy the benefit of fine-grained memory copies, but the data transfer time and the overhead of fine-grained tracking are neglected. Not attainable in practice, this study help us understand how close each multi-GPU programming paradigm comes to a theoretical maximum performance. We compute this performance bound by using the bulk transfer implementation of each workload and then discounting its execution by the time spend performing data copies with `cudaMemcpy`.

2.4.3 Benchmarks

To evaluate PROACT we implement multiple versions of workloads across a variety of scientific domains and microbenchmarks that can demonstrate the differences between the PROACT transfer mechanisms. All our benchmarks are compiled using CUDA 9.1 [111].

Microbenchmarks: For demonstrating system variation, we use microbenchmarks to compare the different PROACT design points. These microbenchmarks consist of a

synthetic compute kernel running on a *source GPU*, which generates data that is needed in entirety by the *destination GPUs* for the next phase. Because the opportunity to overlap data transfer and kernel execution is maximized when the duration of the compute kernel and transfer are equal, we tune the compute time of the synthetic kernel to match the transfer time under `cudaMemcpy()` transfers. To obtain a reasonable size of the compute kernel that can meet this criteria, we fix the total amount of data to be transferred at 256MB and tune the compute across various GPU generations. We then instrument the compute kernel on the source GPU to track its data production and initiate transfers via both PROACT's decoupled mechanisms.

MBIR X-ray CT: Model Based Iterative Reconstruction (MBIR) is a computational technique that can produce high quality images with low X-ray dose, but at high computational cost. MBIR can be formulated to admit considerable parallelism over image voxels or detector values, making it well-suited for multi-GPU acceleration [81, 87, 134]. The algorithm operates iteratively in two barrier-synchronized phases wherein data structures are partitioned across GPUs and each GPU requires input data from the previous phase from all other GPUs. We study an algorithm similar to that used in the FDA-approved GE Veo CT system, the only MBIR system approved for clinical use.

Page Rank: Page Rank assigns a 'Page-rank score' to web pages based on their importance. Our benchmark assigns page-rank scores to articles in the Wikipedia dataset [35].

Single Source Shortest Path (SSSP): Shortest Path algorithms are used to navigate between physical locations, such as in a road network [53]. We use a version of this algorithm where the path computation is formulated as a matrix operation [66]. In each iteration, every vertex computes its shortest distance from the source vertex using the Bellman-Ford algorithm [66]. We compute SSSP on the HV15R dataset [35].

Alternating Least Squares (ALS): ALS is widely used to perform matrix factorization in recommender systems. The algorithm is iterative. In each iteration, it fixes the user matrix and optimizes the item matrix and vice versa. In our study, we performed ALS using

Stochastic Gradient Descent for vertices in the HV15R dataset [35].

Jacobi Solver: The Jacobi Solver iteratively solves a system of linear equations of the form $Ax = b$, where A is the coefficient matrix, b is a constant vector and x is the required solution vector. The solver iteratively computes the solution vector using Jacobi's method [165]. We performed our study on the Jacobi solver for banded matrices, which arise widely in finite element analysis [164, 70].

2.5 Results

We first analyze PROACT's decoupled transfer mechanisms to understand how they manifest across different GPU architectures and then present PROACT's scalability results.

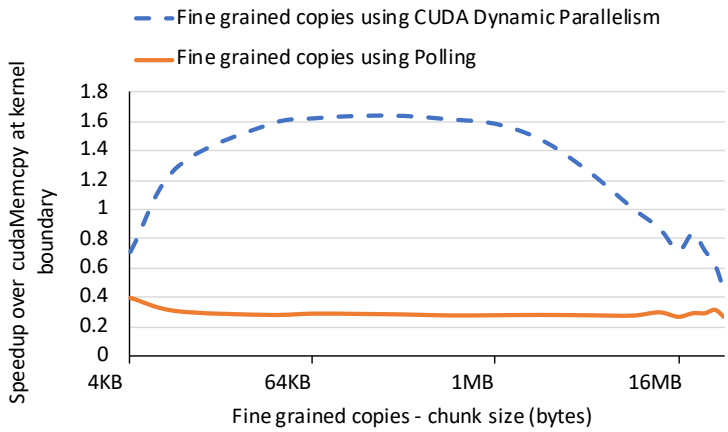
2.5.1 Microbenchmarking Decoupled Transfer Mechanisms

Figure 2.6 shows the performance of the microbenchmark described in Section 2.4.3 when varying the granularity for decoupled transfers from 4KB to 256MB, where each source thread block generates 4KB of data.

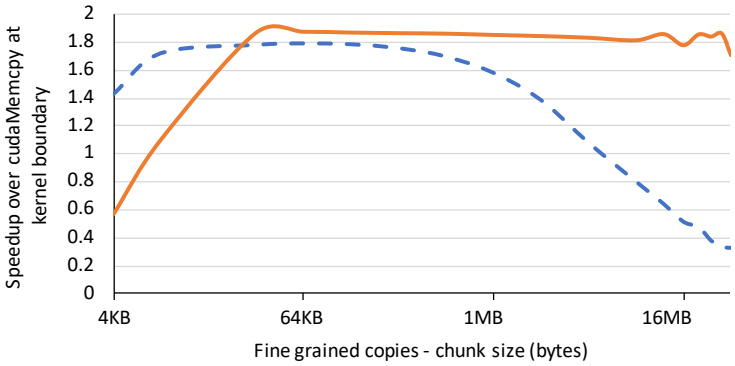
For the Kepler-based system and CDP case, the performance curve exhibits three regions. When the transfers happen at fine granularity (less than 16KB per chunk), performance is *initiation-bound*. Initiation overheads dominate and there is a net slowdown relative to `cudaMemcpy`. From 16KB to 1MB per chunk, the transfers are *bandwidth-bound* and proactive transfers reach their peak speedup of 1.6×.

As the granularity increases beyond 1MB, we enter a *tail-transfer–bound* region, where the left-over transfers, after the compute kernel completes (which we call the tail transfers) become significant leading to a net slowdown. Polling substantially underperforms both `cudaMemcpy` and CDP on Kepler, due to GPU resources wasted by numerous fruitless poll loops.

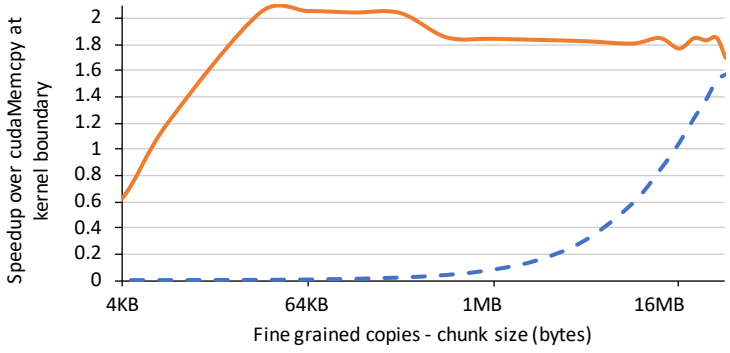
For the Pascal-based system, CDP matches the trend of the Kepler-based system, offering a peak speedup of 1.8× in the bandwidth bound region.



(a) 4x Kepler



(b) 4x Pascal



(c) 4x Volta

Figure 2.6: Performance of the PROACT microbenchmarks showing the effect of decoupled transfer paradigm and transfer granularity.

However, polling performs even better, attaining up to 1.9× speedup once the transfer granularity is large enough so that the delay of iterating over the polling bitmap is amortized

over the transfer operations.

In the Volta-based system, CDP results in slowdowns at low granularities, while achieving a 1.5x speedup at large granularities. The initiation overhead of dynamic kernels is higher on Volta than the other architectures. Polling offers higher speedups at nearly all granularities.

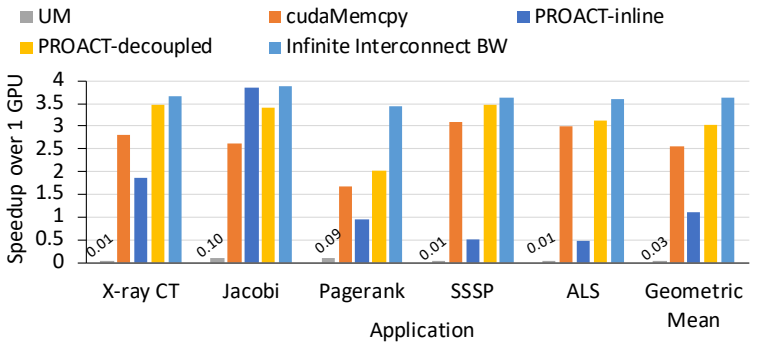
On current GPUs, launching dynamic kernels requires intervention from the host driver. As such, the degree to which dynamic kernel launches impacts the performance of the parent depends on the GPU hardware and the driver. We have observed this cost to vary substantially across GPU generations. Furthermore, the degree the data transfer kernel interferes with the parent computation depends on the number of warps in the data transfer kernel and again varies across GPU platforms. Thus, per GPU platform, the highest-performing data transfer mechanism varies across transfer granularity, transfer mechanism (DMA vs fine-grained) and transfer agent type (polling vs CUDA dynamic parallelism)

2.5.2 End-to-End Performance on Full Applications

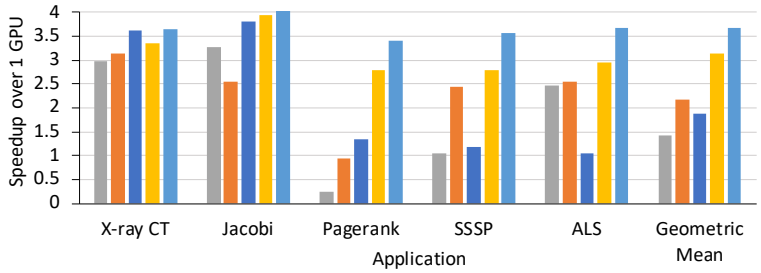
Figure 2.7 shows the speedups achievable on a 4-GPU system over a single GPU, across three different GPU generations and Table 2.2 depicts the corresponding PROACT configurations.

With programmer-provided `cudaMemAdvise` hints, UM can substantially improve performance over a single GPU for some applications. For Jacobi, UM even outperforms `cudaMemcpy` duplication. Performing explicit duplication via `cudaMemcpy` on these 4-GPU systems outperforms a single GPU for all applications except Pagerank, where it underperforms a single GPU on our Volta- and Pascal-based systems.

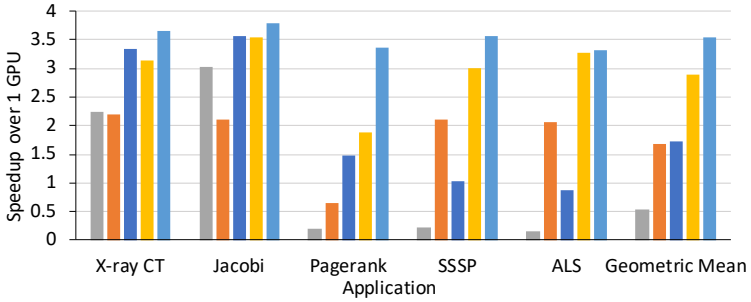
PROACT-inline outperforms decoupled transfers in just four cases where the applications naturally exhibit spatial locality. In Jacobi and X Ray-CT, GPU threads are scheduled so that data are produced densely in increasing address order, leading to excellent write coalescing within the SM. In these cases the increase in interconnect efficiency achieved by decoupling



(a) 4x Kepler



(b) 4x Pascal



(c) 4x Volta

Figure 2.7: 4-GPU speedup under each data transfer method for different hardware configurations.

data transfers is not sufficient to overcome the overheads of the SW-implementation consuming SM resources. This marginal difference in performance is an artifact of prototyping PROACT-decoupled in software and we expect a hardware implementation to outperform the inline configuration in all cases.

In the remaining applications, where coalescing is poor due to more random memory update ordering, PROACT-decoupled performs best, as its larger-grain transfers always coalesce. For example, in ALS on 4x Volta, there are 26× more store transactions over

Application	4x Kepler	4x Pascal	4x Volta
X-ray CT	D 16kB 256 DKL	I	I
Jacobi	I	I	D 128kB 2048 PST
Pagerank	D 16kB 256 DKL	D 1MB 4096 PST	D 128kB 2048 PST
SSSP	D 16kB 256 DKL	D 1MB 4096 PST	D 128kB 2048 PST
ALS	D 16kB 256 DKL	D 1MB 4096 PST	D 128kB 2048 PST

Table 2.2: Best performing PROACT configuration as determined by the PROACT profiler. Each configuration is represented as transfer scheme (I: PROACT-inline and D: PROACT-decoupled), transfer granularity, number of transfer threads, decoupled transfer mechanism (PST: Persistent Software Threads and DKL: Dynamic Kernel Launches).

the interconnect under PROACT-inline than PROACT-decoupled due to poor coalescing. Depending on the nature of the application and system architecture, PROACT picks the most suitable mechanism to perform the data transfers.

The abundant parallelism available in these applications is apparent in the near-linear scaling achieved under the theoretical upper bound with infinite interconnect bandwidth. Regardless of GPU generation, the geometric mean theoretical opportunity is a 3.6× speedup over a single GPU for this set of applications. PROACT enables 3× speedup across all generations, capturing 83% of the possible opportunity. The story is more complicated for `cudaMemcpy`, where there is significantly more variation across GPU generations, but with an average of 2.1× speedup over a single GPU. UM displays the most variable results, achieving good speedups for some applications, on some platforms, but on average underperforming even a single-GPU configuration. We conclude that, even with programmer-directed hints, traditional fault-based UM can not be relied upon to achieve peak performance. The minimal additional overhead for programmers to use PROACT is a small price to pay to achieve the highest performance across applications and GPU systems.

2.5.3 Decomposing PROACT's Performance

To further understand the performance, we analyze the overheads incurred by PROACT. PROACT-inline incurs no overhead on computation since there is no tracking instrumentation involved. For PROACT-decoupled, the largest performance overhead stems from

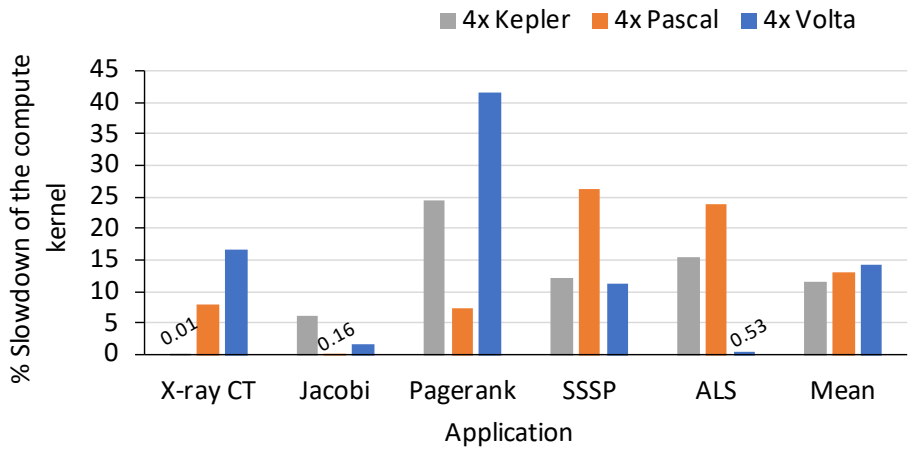


Figure 2.8: Compute slowdown due to PROACT for decoupled transfers (included in all our results).

prototyping the data tracking interface entirely in software. We determine the overhead by comparing runtime with tracking instrumentation but without the data transfers to the runtime of the theoretical infinite interconnect BW case. We report this overhead in Figure 2.8.

The figure reveals that the overhead averages between 10 and 15% depending on GPU platform. The variation across applications is significant, ranging from negligible overhead to as much as 40% for Pagerank. Note that the overhead is included in all our results (both in Figure 2.7 and the Figure 2.10), which shows that PROACT-decoupled achieves substantial performance improvements despite the software overhead. A hardware implementation will alleviate these overheads and can further improve performance, motivating its inclusion in future GPU architectures.

To understand how efficient PROACT is at overlapping data transfer and computation, we investigate the fraction of data transfer overhead overlapped with computation on each of the 4-GPU systems. We obtain this result by executing each application with the instrumentation and initiation overheads of PROACT, but eliding the stores that actually perform the data transfers to remote GPU memory. The difference between the runtime with and without the data transfer operations reveals the portion of transfer time that is not overlapped. We then determine the fraction of overlap by comparing the non-overlapped transfer time to the baseline duplication time with `cudaMemcpy`, reported in Figure 2.9.

Although there are variations among applications and platforms, Figure 2.9 reveals that PROACT always hides at least 75% of transfer time. In many cases it can overlap nearly 100% of the communication, which will enable great scalability at high GPU counts relative to `cudaMemcpy` duplication.

2.5.4 Strong Scaling with PROACT

To this point we have shown studies of PROACT's (inline and decoupled) performance impact for 4-GPU systems with different generations of GPUs and interconnects. We next consider the scalability of PROACT with multi-GPU system size. Figure 2.10 shows the absolute speedup that PROACT achieves on Kepler, Pascal, and Volta for systems with up to 4, 4, and 16 GPUs, respectively.

In Figure 2.10 we omit unified memory results, which do not scale well (on average) and instead focus on the performance improvements with PROACT and `cudaMemcpy` duplication compared to the single-GPU performance achieved on the respective system. We see that across all three platforms, when employing just two GPUs performance is insensitive to transfer method. This result is not surprising; as the total fraction of time spent in data transfers is insignificant with only two GPUs. However, as the number of GPUs increases to four (on the Kepler- and Pascal-based systems), and 5–6 on the Volta-based system, performance with `cudaMemcpy` flattens and even begins decreasing.

This effect manifests beyond just two GPUs on the Kepler system, which we hypothesize is because our PCIe-based Kepler system has the lowest inter-GPU bandwidth, thus transfer overheads start affecting performance more quickly than in the other systems. We see that, on the Pascal system, performance with `cudaMemcpy` is generally competitive up to three GPUs before leveling off. When examining many-GPU scaling on the Volta-based prototype system, `cudaMemcpy` scales to five GPUs before leveling off. PROACT, however, exhibits nearly linear multi-GPU scaling, indicating that it is doing a good job using the interconnect bandwidth effectively and overlapping GPU communication with computation. Overall, on

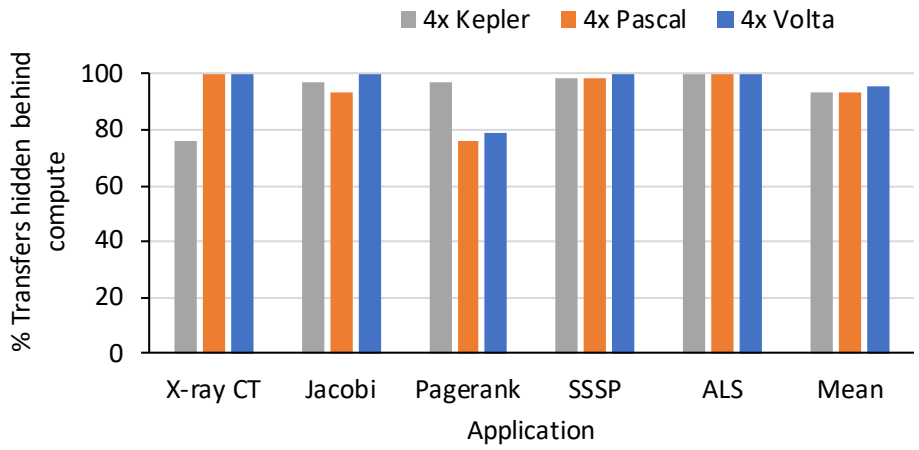


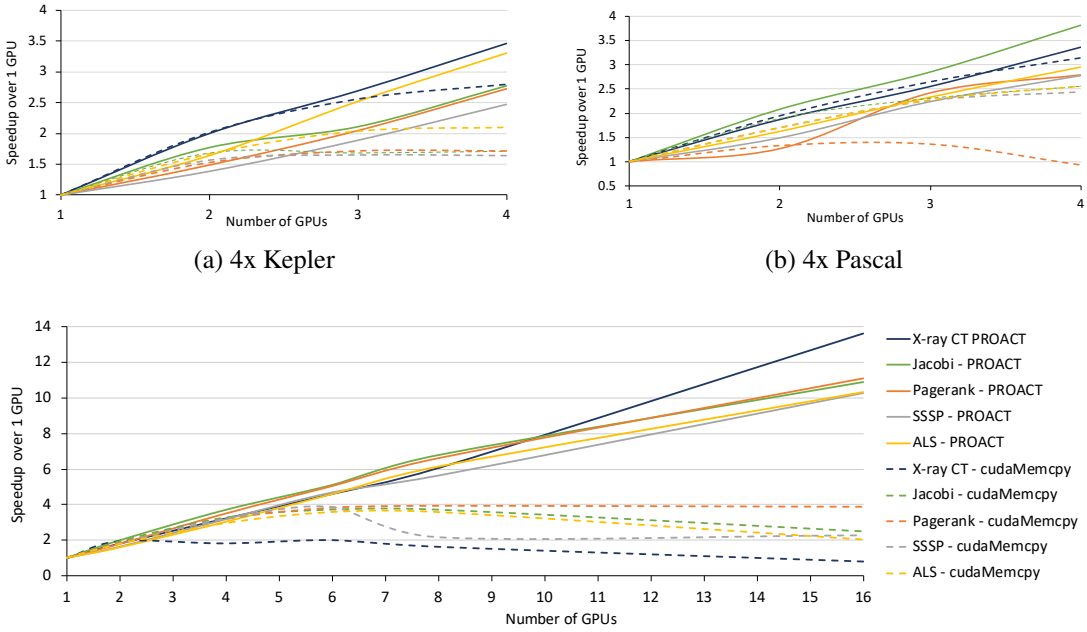
Figure 2.9: Transfer overlap achieved by PROACT.

our 16 GPU Volta system PROACT achieves a mean speedup of 1.2×, 2.2× and 5.3× over `cudaMemcpy` duplication at 4, 8, and 16 GPU configurations, respectively, while coming within 90%, 87% and 77% of the theoretical application performance limit.

2.6 Related Work

Accelerating scientific applications using GPUs [44, 168, 43, 20, 153, 30, 135] and optimizing GPU communication [158, 125, 122] has been widely studied, but in what are now legacy contexts; given the introduction of direct switch-connected LD/ST-based multi-GPU systems.

Communication in GPU systems: To our knowledge, ours is the first work to explore proactive direct stores to optimize multi-GPU communication. Xiao et al. [167] optimize inter-block GPU communication via barrier synchronization. CudaDMA [13] overlaps computation and data transfer between GPU global and shared memory. MVAPICH2-GPU [158] integrates CUDA data movement transparently with MPI. Potluri et al. provide intra-node MPI communication on multi-GPU nodes in [125]. CGCM [56], is an automated system for managing and optimizing CPU-GPU communication. Chen et al. [27] explore weak execution ordering to reduce host synchronization overhead.



(a) 4x Kepler

(b) 4x Pascal

(c) 16x Volta

Figure 2.10: Scalability achieved on different hardware configurations.

Groute [16] provides an asynchronous multi-GPU model for irregular applications; but does not consider optimizing fine grained communications between GPUs. Even in MCM-GPU [7], where a package-level integration of multiple GPU modules is done, PROACT could help improve performance by effecting efficient copies to local DRAM partitions. Prior work [89, 171, 11, 68] has explored various hardware and software mechanisms to improve multi-GPU performance. Many other works [173, 1, 75, 128, 38] perform NUMA-aware optimizations to improve GPU performance and perform hardware-based peer caching [83, 126, 31, 143, 12].

Performance of heterogeneous systems: Staged Memory Scheduling [8] decouples the primary tasks of a memory controller to improve performance of CPU-GPU systems. Sutherland et al. [149] improve performance by using texture cache approximation on GPUs. While Schaa et al. [137] allow developers to accurately predict execution time of GPU scaling, Yao et al. [170] provide a theoretical analysis of multicore scalability. Unicorn [18] provides a parallel programming model for hybrid CPU-GPU clusters while

[22, 15] provide frameworks for automatic multi-GPU parallelization. CAWS [74] provides a scheduling policy and Bhattacharjee et al. propose thread criticality predictors for parallel applications [19]. Dymaxion [25] attempts to overlap CPU-GPU PCIe transfer with data layout transformations on the GPU, while Lustig et al. [83] offer techniques for more effective GPU offloading. These topics are orthogonal and complimentary to optimizing fine-grained multi-GPU communication, the subject of this work.

Auto-tuning and code generation: General literature in auto-tuners include automating the construction of compiler heuristics using machine learning [146], automatic tuning of heuristics for code inlining [24], and a genetic algorithm approach for compiler optimizations [71]. Additionally, obtaining the heuristic scheduling algorithm automatically [92], and finding the shortest program to compute a function [85] have been studied previously. In the GPU realm, multiple works have explored application-specific auto-tuning for GPUs [104, 79, 65, 33, 34, 78, 46], but this work provide a new methodology for scaling multi-GPU performance. Spafford et al. attempt to improve load balancing and bus utilization [147], yet they do not attempt to overlap compute with communication. CLTune [103] provides a generic auto-tuner for OpenCL Kernels but does not consider inter-device communication. While different works have explored code generation for specific GPU applications [51, 155, 29, 4, 174], we attempt to create a generic mechanism allowing all applications to scale.

2.7 Conclusion

In this work, we proposed PROACT, a hardware and software system that improves multi-GPU performance by overcoming the limitations of existing inter-GPU communication mechanisms. PROACT combines the advantages of peer-to-peer transfers with that of bulk transfers to enable interconnect friendly data transfers while hiding transfer latencies. Demonstrated across three different 4-GPU system architectures, PROACT provides an average speedup of 3$\times$ over a single-GPU implementation, capturing 83% of the theoretical

limit. We also show how PROACT provides dramatic scalability improvements on next generation systems with large numbers of GPUs, achieving up to 12× speedup over a single-GPU implementation, while capturing 77% of the available opportunity. Scalable multi-GPU programming is no easy task; to maximize programmer productivity, runtime systems like PROACT will be necessary to enable rapid development cycles while leveraging next generation architectural improvements in future GPUs.

CHAPTER III

Case Study: Improving GPU Scaling for X-Ray CT*

3.1 Introduction

Model-based iterative reconstruction (MBIR) for X-Ray CT offers improved image quality at lower radiation doses than Filtered Back Projection (FBP) [152], but at higher computational costs. Although researchers have explored various acceleration techniques, such as using SIMD instructions on CPUs [172, 136] and cloud computing [100, 133], the compute times remain undesirably high for MBIR to be ubiquitous clinically.

Further MBIR acceleration requires increasing both computational resources and memory bandwidth, making a case for employing multiple GPUs [86, 57]. But GPU scaling does not always result in linear speedup. Fig. 3.1 shows projected speedups as we parallelize a state-of-the-art MBIR algorithm [67] over more GPUs. Using more GPUs initially provides near-linear speedup up to about eight GPUs, as the computational phases of MBIR can be readily partitioned across the GPUs. However, beyond eight GPUS, speedup saturates and then begins to decrease. This reversal arises due to the time spent copying data among GPUs between computational phases. Sinogram and image data must be exchanged all-to-all among the GPUs between phases, yet current systems offer no mechanism to broadcast data, requiring pairwise copies. As a result, even though computation time shrinks, copy time grows and ultimately dominates as the number of GPUs increases.

*This chapter based on [98]

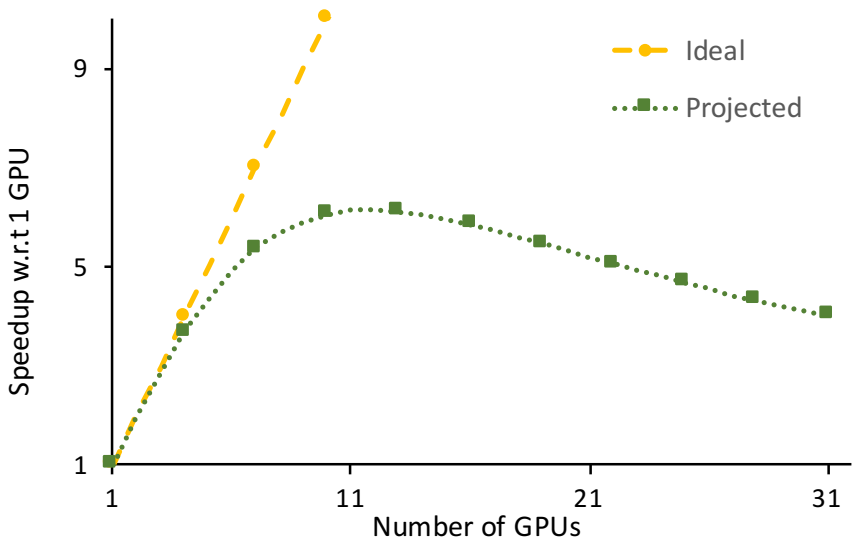


Figure 3.1: Ideal and Projected speedups achievable with GPU scaling with conventional multi-GPU implementation.

High copy time can be mitigated by overlapping copying with computation—by "streaming" data from producer GPUs as soon they become ready. By initiating some copies while computation is ongoing, the next computational phase must wait only for straggling data produced at the end of a phase.

While overlapping communication with compute between CPU and GPU has been studied for FBP[176], hiding copies underneath compute in multi-GPU systems entails several challenges: (1) State-of-the-art CPU–GPU systems provide an astonishing diversity of mechanisms to move data from one GPU's memory space to another. Data may be moved by CPU loads/stores through a variety of addressing mechanisms, by GPU loads/stores, by DMA engines integrated on the GPU, and potentially even by other devices on the PCIe bus. These alternatives trade off bandwidth, initiation latency, and disruption to other GPU threads in non-trivial ways; the best approach for our purpose is not obvious. (2) Every CT phase must indicate when enough data has been generated for a copy to start. It is neither clear how to trigger copies, nor at what granularity they should be performed. (3) The CUDA programming model allows enormous freedom in ordering the execution of individual threads. No existing programming interface allows GPU programs to efficiently

track production of output data, initiate copies, and await copy completion.

This work focuses on using PROACT for accelerating a penalized weighted least-squares with ordered subsets (PWLS-OS) reconstruction algorithm [67] on a multi-GPU system. PROACT automatically identifies the best granularity at which to perform such copies and the transfer mechanism best suited for our purpose. We also consider how the subdivision of MBIR phases into individual GPU kernels affects the order data is generated and how to orchestrate these to maximize gains from our technique.

Using four Tesla K40m GPUs, we show that a 20-iteration helical CT reconstruction of a 512×512×512 image from 7256 views of size 888×64 takes 11.25 minutes using our copy mechanism. We also demonstrate, using a simple mathematical model, how PROACT makes MBIR CT more amenable to further GPU scaling.

3.2 Methods

3.2.1 Background

We reconstruct the image $\widehat{\boldsymbol{x}}$ by iteratively minimizing the PWLS cost function [133]:

$$\widehat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}\geq 0} \Psi(\boldsymbol{x}), \qquad \Psi(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}\|^2_{\boldsymbol{W}} + \boldsymbol{R}(\boldsymbol{x}), \tag{3.1}$$

where $\boldsymbol{A}$ is the system matrix, $\boldsymbol{y}$ is the sinogram measurements, $\boldsymbol{W}$ is the statistical weighting and $\boldsymbol{R}$ is the regularizer.

Each iteration updates the current image estimate ($\boldsymbol{x}^{(n)}$) using the following gradient of Ψ:

$$\nabla\Psi(\boldsymbol{x}^{(n)}) = \boldsymbol{A}^T\boldsymbol{W}(\boldsymbol{A}\boldsymbol{x}^{(n)}-\boldsymbol{y}) + \nabla\boldsymbol{R}(\boldsymbol{x}^{(n)}). \tag{3.2}$$

Each iteration comprises four phases as shown in Figure 3.4. Each phase can be formulated to admit considerable parallelism over image voxels or detector values in the sinogram, making it well-suited to using the enormous compute capabilities of modern GPUs [93].

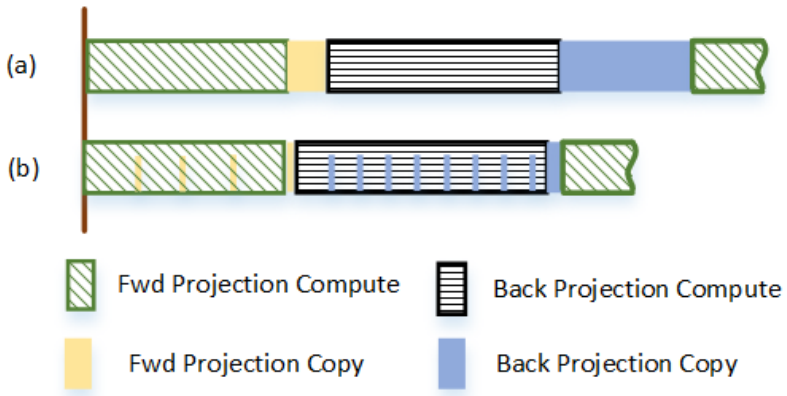


Figure 3.2: (a) Conventional: computation and copies in series. (b) PROACT: incremental copying overlapping computation

GPUs have multiple Streaming Multiprocessors that concurrently execute many threads [80]. These hardware elements execute a GPU kernel (programmed in CUDA) organized as blocks, warps and threads [69]. Kernel ordering is programmer-controlled, but ordering of blocks and warps is left to the hardware scheduler.

In our GPU implementation of PWLS-OS, we parallelize forward projection by partitioning views across multiple GPUs, employing one GPU thread to compute the value of one detector residual. Every GPU broadcasts its generated residual values to every other GPU, since back projection uses all of them. We then parallelize computation of the other phases by partitioning the y plane across multiple GPUs. Each thread performs the back projection, regularization and update of one image voxel. The GPUs then copy the corresponding partial image to other GPUs before the next iteration commences.

3.2.2 Using PROACT to overlap copy with compute

As shown in Figure 3.1, although PWLS-OS is highly amenable to parallelization, GPU scaling does not yield linear speedup due to time taken to perform all-to-all copies of the detector residual at the end of forward projection and of the image voxels at the end of the update phase. The total amount of data copied increases linearly with number of GPUs; although each GPU produces fewer values, they must copy to more destinations. Since

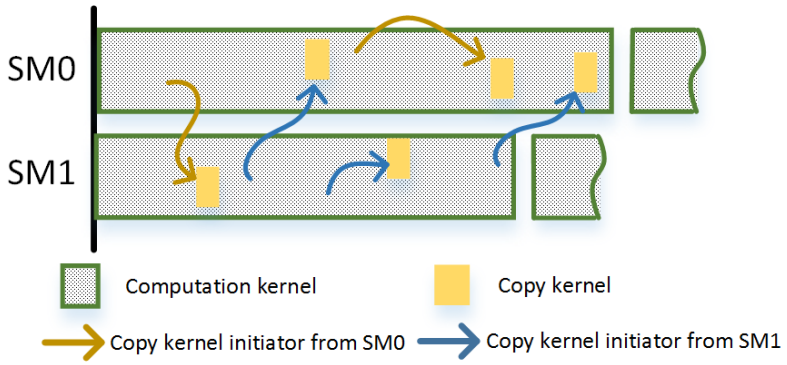


Figure 3.3: GPU SM resource utilization and copy initiation. The launch of the initiated copy kernels depends entirely on the hardware scheduler.

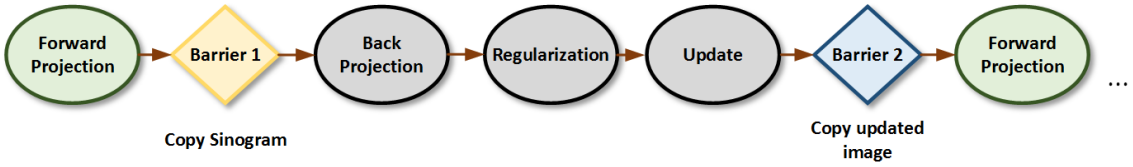


Figure 3.4: Phases of X-Ray CT MBIR. After forward projection, the GPUs exchange their generated sinogram data, followed by three computational phases and an image update. The GPUs then exchange image data and the algorithm repeats until convergence.

using more GPUs also reduces compute time per GPU, scaling beyond 12 GPUs results in copy time that exceeds compute time.

The available copy bandwidth is bound by the interconnect (PCIe3.0 and PCIe4.0 pose theoretical limits of 16GB/s and 32GB/s, respectively, though substantially lower sustained throughput is achievable in practice) and cannot be increased without hardware enhancements. However, a careful study of the hardware utilization pattern during the reconstruction process indicates that the interconnect remains idle during the computation phases and is utilized to its practical limit only during the ensuing copy phase, when no compute takes place. Hence, one way to decrease the reconstruction time is to use PROACT to initiate copies of data generated early in the computation phase while the rest is still being computed, overlapping computation and copying, as depicted in Figure 3.2.

To overlap compute and copy, the copy mechanism must impose minimal overhead and interference on the computation. PROACT identifies GPU thread-based copy using dynamic kernel launch as the mechanism best suited for our purpose. In this technique, the

GPU threads that perform computation also trigger copies after generating a data chunk (e.g., a set of sinogram bins) of the desired granularity. The copies are issued via *dynamic kernel launch*, wherein a copy kernel is launched from within the main CT computation kernels—a capability introduced in CUDA 5.0 [62]. The copy kernel is scheduled by the GPU hardware scheduler and copies the data chunk to the other GPUs as shown in Figure 3.3. As computation proceeds, copy kernels are triggered, scheduled by hardware, and executed.

Although our technique incurs minimal copy initiation overhead, the copy kernels nevertheless use GPU resources that might otherwise have been used by CT computation kernels, and hence indirectly delay computation. Hence, invoking the copy kernel at an appropriate frequency while ensuring that the maximum amount of copy time is hidden behind compute becomes crucial. PROACT addresses this challenge by carefully tuning the granularity at which copies are initiated. Since this granularity depends only on the CT geometry and not on patient features, it can be determined in advance via a parameter-sweep over a sample image.

Choosing the right granularity to track data production is also important, as tracking at too low granularity (such as thread granularity) would increase the overhead of tracking, slowing computation, while tracking at too high granularity might not ensure enough compute-copy overlap. PROACT performs the tracking at the granularity of GPU thread blocks. Once consecutive thread blocks produce a chunk of data of the desired granularity, a dynamic copy kernel initiates the copies.

Our final design decision entails selecting the best implementation to track data generation. Proper design of this mechanism is crucial because the CUDA programming model offers enormous freedom to the scheduler with respect to block ordering, so blocks may complete in any order. PROACT employs an atomic counter-based approach, wherein each data chunk is assigned a corresponding atomic counter, initialized to the number of blocks that contribute data to the chunk. The first thread of each block waits until all its sibling threads complete, then decrements the counter using an atomic decrement instruction. When

the counter reaches zero, it indicates that the corresponding group of consecutive blocks is complete. The thread then initiates a dynamic copy kernel for the corresponding data chunk. Although the counter-based approach uses atomic accesses that are inherently slower than normal reads and writes, it performed better than alternatives (e.g., dedicated threads that poll for chunk completion).

3.2.3 Sequencing data generation

To ensure that the reconstruction method can effectively use the copy strategy discussed above, it is important to structure the kernels to perform all the computation corresponding to a particular data element in quick succession, producing data elements incrementally rather than performing multiple updates to all data elements during kernel execution. The goal is to ensure that data chunks are available as early as possible to maximize copy-compute overlap. While the original forward projection code computed the detector residuals in succession, the back projection code updated the relevant voxels using one view before updating the voxels using the next view, i.e., an outer loop over views. This led to voxels being ready for copy only during the processing of the last view, leaving very little room for compute-copy overlap. We restructured the code to generate the voxels in succession by having each voxel loop over the relevant views that contribute to it, thus ensuring voxel values are produced incrementally.

3.3 Experimental Results

We report on our multi-GPU PWLS-OS implementation. We validate the GPU implementation against a CPU baseline, which it matches to within 0.0289HU (Hounsfield units). Our test system comprises four Tesla K40m GPUs, each having 2880 CUDA cores, 11.9GB global memory and 4KB shared memory per block. Each GPU is capable of performing a peer access to the other GPUs. They reside on a PCIe3.0 bus that also interfaces them with the host.

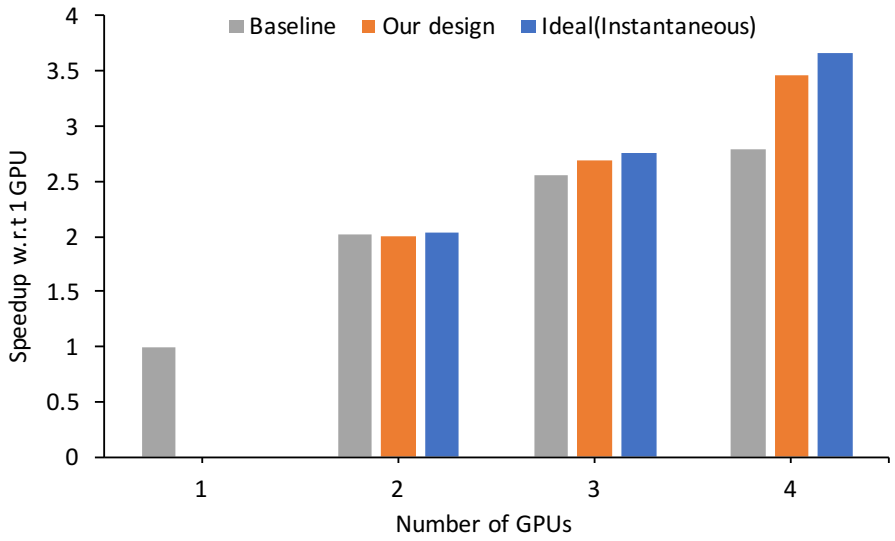


Figure 3.5: Measured speedups achieved through PROACT

We simulated 9-turn helical CT data with pitch 63/64 and 7256 views of size 64 rows by 888 channels, and reconstructed a 512×512×512 voxel image volume over a 512 mm transaxial field of view (FOV) with 0.625 mm slice thickness using the separable footprint projector [82] and [67] with 24 subsets.

In PROACT, the GPU threads initiated through dynamic kernel launch perform copies, as explained in Section 3.2.2. The copies are performed for 8kB data chunks for detector residuals and 16kB for image voxels. We compare PROACT against the baseline wherein data is copied after compute phases using cudaMemcpy Peer-to-Peer [138].

Figure 3.5 shows speedup achieved with respect to a single GPU for three cases: (1) Baseline, (2) Our design (PROACT), and (3) Instantaneous copies (copies are performed in zero time). The blue (Instantaneous) bars indicate the theoretical limit on the performance gains achievable via compute-copy overlap. With only two GPUs, copy overhead is negligible and there is little performance difference between the baseline and the ideal. However, the potential and realized gains grow rapidly with further scaling. Our approach realizes 94% of the theoretical opportunity for four GPUs, achieving a speedup of 1.24× over the baseline and 3.46× over a single GPU.

PROACT falls short of the opportunity available with instantaneous copies for two

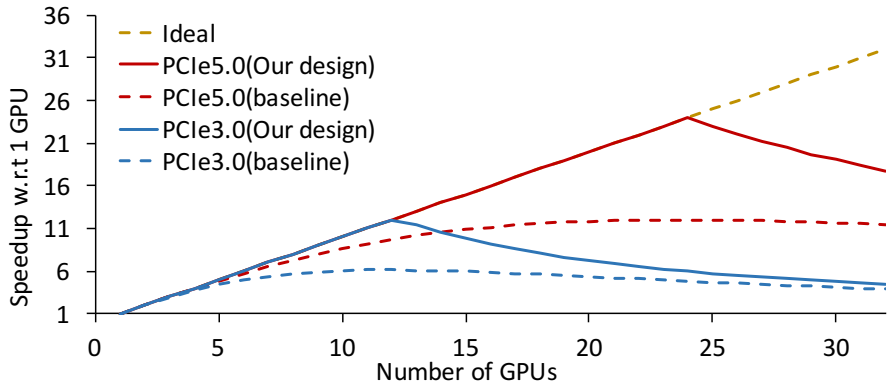


Figure 3.6: Effect of GPU scaling on speedup for different interconnect generations

reasons: (1) The dynamic copy kernels require some GPU execution bandwidth, slightly delaying execution of CT kernel threads. (2) A 100% copy-compute overlap is not possible because the data generated by the final blocks is copied after computation is complete.

3.4 Scalability Analysis

PROACT holds the potential to unlock even higher performance scalability on future, larger multi-GPU systems. To analyze this potential, we develop a simple analytic model that predicts the impact of our approach with more GPUs and faster GPU interconnects. We base our scalability model on the following observations: (1) As previously explained, the four phases of X-ray CT are amenable to GPU scaling. For simplicity, we assume that the compute time of individual algorithmic phases scales linearly with the number of GPUs, although practical implementations typically fall a bit short of ideal linear scaling. (2) An all-to-all broadcast must occur at the end of the forward projection and update phases. Thus, as the number of GPUs increases, the total data to be copied, and hence the time required for copy, increases, since the data must be copied to additional GPUs.

3.4.1 Scalability Model

At the end of forward projection, the GPUs must exchange the portions of the sinogram each generated. The total bytes copied is the product of the sinogram size ($Size_{sino}$), and

the number of destination GPUs ($n_{\text{gpu}} - 1$). The total time for the copy depends upon the interconnect technology, which we model simply as a 'copy time per byte (t_{perbyte})' bandwidth, as expressed in the following equation:

$$\text{Copytime}_1 = (n_{\text{gpu}} - 1) * \text{Size}_{\text{sino}} * t_{\text{perbyte}}.$$

After the update phase, the GPUs must exchange the image portions each generated. Hence the total bytes copied is the product of the total image size ($\text{Size}_{\text{image}}$) and the number of destination GPUs ($n_{\text{gpu}} - 1$):

$$\text{Copytime}_2 = (n_{\text{gpu}} - 1) * \text{Size}_{\text{image}} * t_{\text{perbyte}}.$$

3.4.2 Discussion

Using this simple model, we estimate the impact of PROACT on CT reconstruction performance. Figure 3.6 shows the projected speedup of PROACT and that of the baseline for different GPU counts against a single GPU implementation for two different assumptions on interconnect bandwidth (unidirectional transfer bandwidths of 16GB/sec for PCIe 3.0 and an estimated 64GB/s for PCIe 5.0).

For both assumptions on interconnect bandwidth, PROACT enables performance scalability to a much larger number of GPUs than the baseline. For PCIe 3.0, baseline performance saturates at about 6× speedup (over a single GPU) with ten GPUs. Above six GPUs, growth in copy time exceeds reductions in computation time. In contrast, our approach enables near-ideal scaling up to twelve GPUs. Beyond this point, copy time grows to the point where it exceeds compute time and can no longer be hidden.

Higher interconnect bandwidth under PCIe 5.0 reduces copy time, enabling greater performance scalability for both the baseline and PROACT. However, our technique still drastically increases the scalability potential. PROACT enables near-ideal scaling up to 24

GPUs, while the baseline saturates at about 12× speedup with 20 GPUs. Newer GPUs will further decrease compute time by the time PCIe 5.0 becomes available, making our solution even more relevant.

3.5 Summary and Conclusion

In this Chapter, we showed that though Model-based iterative reconstruction (MBIR) for X-Ray CT is highly parallelizable, reconstruction time does not improve linearly with the number of GPUs, mainly due to high inter-GPU copying delays at the end of computation phases. We also showed that using PROACT to overlap copies with computation—by copying incrementally as data are produced—can mitigate copy overhead and improve performance scalability. PROACT enabled 90% of copy time to overlap with compute, achieving a speedup of 1.24× (nearing the theoretical bound of 1.31× with instantaneous copies) over conventional cudaMemcpy at the end of compute phases on four Tesla K40m GPUs. Relative to a baseline implementation on a single GPU, PROACT approach achieves a speedup of 3.46× on four GPUs. We project even higher impact from our technique with more GPUs.

CHAPTER IV

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

4.1 Introduction

GPUs are now central to high performance computing, given their abundant resources and suitability for parallel execution. However, extracting high performance from multi-GPU systems remains a challenge. Multi-GPU systems offer teraflops of computational power [117, 47] and terabytes per second of memory bandwidth. However, effectively managing these resources to extract high performance from multi-GPU systems remains a challenge for most GPU users. As the GPU count in a system increases, the time each GPU spends on computation decreases while inter-GPU communication increases, resulting in poor performance.

One of the primary challenges for effective multi-GPU memory management is the order of magnitude gap between local and remote memory bandwidths. If applications are naively partitioned across GPUs, most memory accesses will end up traversing slow remote links, resulting in these links becoming a performance bottleneck. Interconnect bottlenecks can degrade scaling so significantly that multi-GPU implementations often perform worse than their single-GPU counterpart.

*This chapter based on [95]

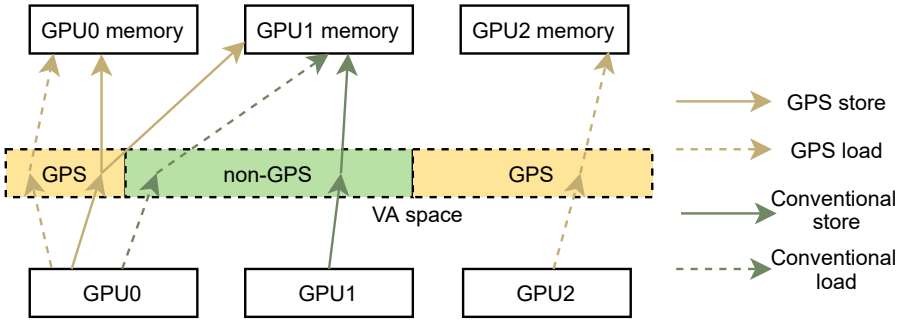


Figure 4.1: Load/store paths for conventional and GPS pages.

4.1.1 Challenges of existing solutions

Existing techniques to manage data in multi-GPU systems fall short. Unified Memory (UM) [48] provides a single memory address space accessible from any processor in the system by employing fault-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. Peer-to-peer transfers [138], 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, orchestrating this requires substantial manual programmer effort and detailed knowledge about the application and execution platform. Frameworks such as Gunrock [159] and Groute [17] can help improve strong scaling for some workloads, but these frameworks are typically domain specific and limited in scope.

4.1.2 Contributions

To overcome the limitations of existing techniques, we propose GPS (GPU Publish Subscribe), a multi-GPU memory management technique that transparently improves the performance of multi-GPU applications following a Unified Memory-like 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 4.1 shows the behavioral difference between GPS and conventional accesses. GPS loads always return from local memory, while GPS stores are issued to all subscribers. 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 interconnect bandwidth efficiently.

GPS successfully improves strong scaling performance because of its ability to: (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 GPS, a multi-GPU memory management technique that leverages the publish-subscribe paradigm.
- We propose simple, intuitive programming model extensions that naturally integrate GPS into applications, while still conforming to the existing GPU memory model.
- We propose a novel subscription management mechanism that tracks GPUs' access patterns and unsubscribes from unused pages to minimize scarce inter-GPU bandwidth.
- We demonstrate that GPS enables strong scaling performance up to 3.84× over a single GPU for a suite of native multi-GPU applications, an average 2×-5× improvement over the next best available multi-GPU paradigm across different interconnect architectures, and achieves 75% of the hypothetical performance of an ideal interconnect.

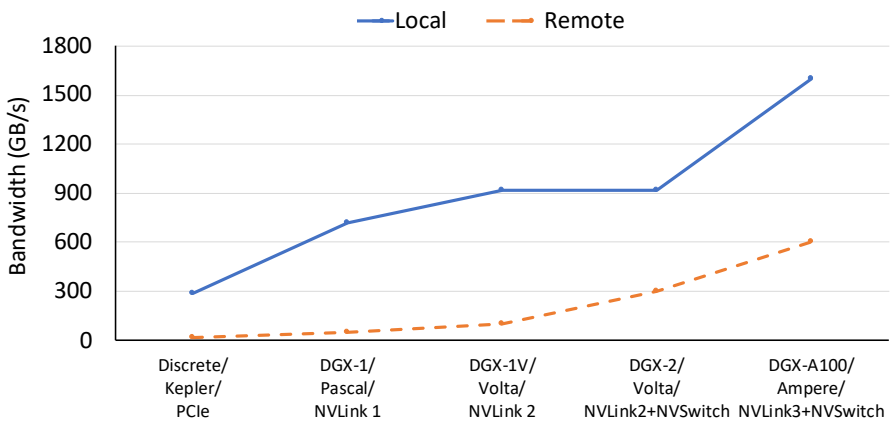


Figure 4.2: Local and remote bandwidths on varying GPU platforms. Despite significant increases in both metrics, a 3× bandwidth gap persists between local and remote memories.

4.2 Background and Motivation

4.2.1 Proactive transfers for locality

Proper orchestration of remote accesses 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.

This trend is illustrated in Figure 4.2 which shows the improvements in local and remote memory bandwidths across GPU and interconnect generations. We see that even though interconnect bandwidth has improved 18× while evolving from PCIe 3.0 [3] to NVIDIA's most recent NVSwitch [115], it still remains 3× slower than the local GPU memory bandwidth. If GPUs perform remote loads via the slower interconnect links during computation, they incur computation stalls and poor performance.

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.

4.2.2 Inter-GPU communication mechanisms

To exploit the advantages of local memory accesses, multi-GPU workloads typically use one of following mechanisms to move data among GPUs' physical memories.

- **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) [123], the MPI library can send and receive GPU buffers directly, without having to first stage them in host memory. Message transfers can also be pipelined with computation. However, porting an application to MPU requires significant programmer effort, and as a result, it is hard to overlap compute and communication effectively by leveraging the pipelining features of MPI.

- **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 [144], 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)**: NVIDIA's Unified Memory provides a single unified virtual memory 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 performance concern.

- **Peer-to-peer transfers**: GPU threads also have the ability to 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 remote accesses, often stalling thread execution beyond the GPU's ability to mitigate those stalls via multi-threading. Peer-to-peer stores, on the other hand, 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.

GPS relies on peer-to-peer stores to perform data transfers as the basis of its performance-scalable implementation.

4.2.3 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 [41, 9, 37, 94].

A simple example of a publish-subscribe framework is shown in Figure 4.3. It consists of publishers who generate data and subscribers who have requested specific updates. The subscribers send their access requests to a publish-subscribe processing unit that tracks subscription information at page granularity. The publisher sends data updates to the publish-subscribe processing unit, which then forwards these updates to subscribers. This mechanism provides the advantage that publishers and subscribers can be decoupled from one another and subscription management is handled entirely by the publish-subscribe processing unit.

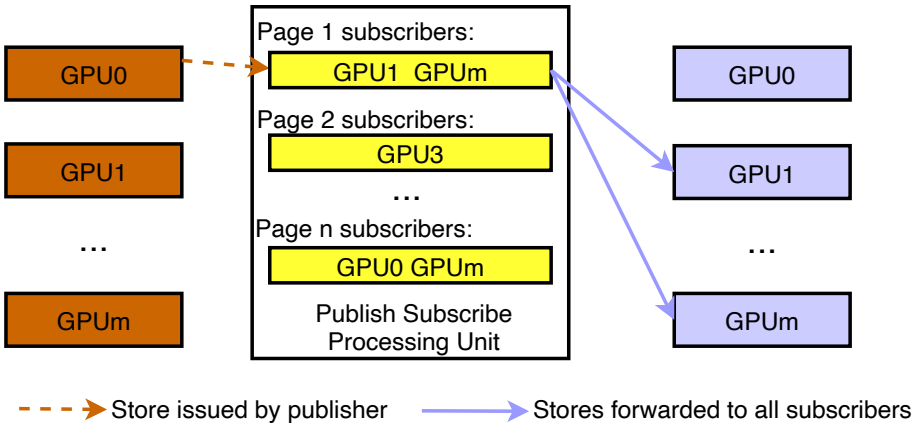


Figure 4.3: A simple publish-subscribe framework.

4.2.4 The GPU memory consistency model

The NVIDIA GPU memory consistency model [118] prescribes rules regarding the apparent ordering of GPU memory operations and the values that may be returned by a read operation. We briefly summarize concepts from the memory model that are relevant to GPS.

Memory operation types: Memory operations can be broadly categorized into two types; namely, *weak* operations and *strong* operations. *Weak* operations are non-synchronizing memory instructions. The effects of these instructions are required to become visible to other threads only when a synchronization is established by an additional explicit event. GPU memory accesses are weak accesses by default, unless they are atomic operations, fence operations, or they possess explicit synchronization qualifiers. *Strong* operations are synchronizing operations, i.e., they are either fence operations or memory operations with synchronization qualifiers.

Scopes: In the GPU memory model, *strong* operations (but not weak operations) have the ability to synchronize at different *scopes* depending on applications' requirements. The GPU thread hierarchy is shown in Figure 4.4. Threads are grouped into Cooperative Thread Arrays (CTAs) a.k.a. thread blocks; multiple CTAs constitute a GPU, and the system consists of all available GPUs. A grid is the one-, two-, or three-dimensional set of CTAs used to execute a particular kernel on a GPU. The GPU memory model provides the freedom to the

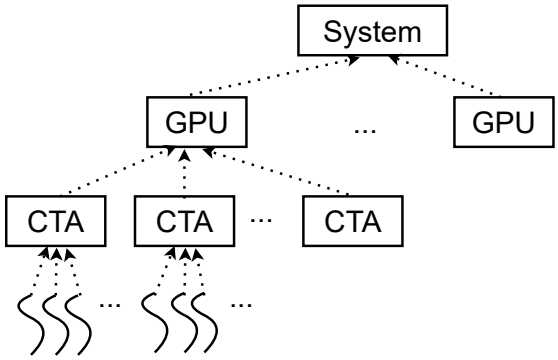


Figure 4.4: The thread hierarchy of NVIDIA GPUs.

user to choose the subset of threads within the thread hierarchy for which synchronization and same-address ordering rules are enforced. The available scopes are:

- `cta` scope, which is the set of all threads executing in the same CTA as the current thread.
- `gpu` scope, which is the set of all threads executing on the same GPU as the current thread.
- `sys` scope, which is the set of all CPU and GPU threads in the system.

GPS makes use of the relaxed memory ordering guarantees offered by the different GPU operation types and scopes to perform several hardware optimizations, as described later in Section 4.3.4.

4.3 GPS Architectural Principles

In this section we describe the different architectural principles upon which GPS is based. Several possible hardware implementations, described later in Section 4.5, 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.

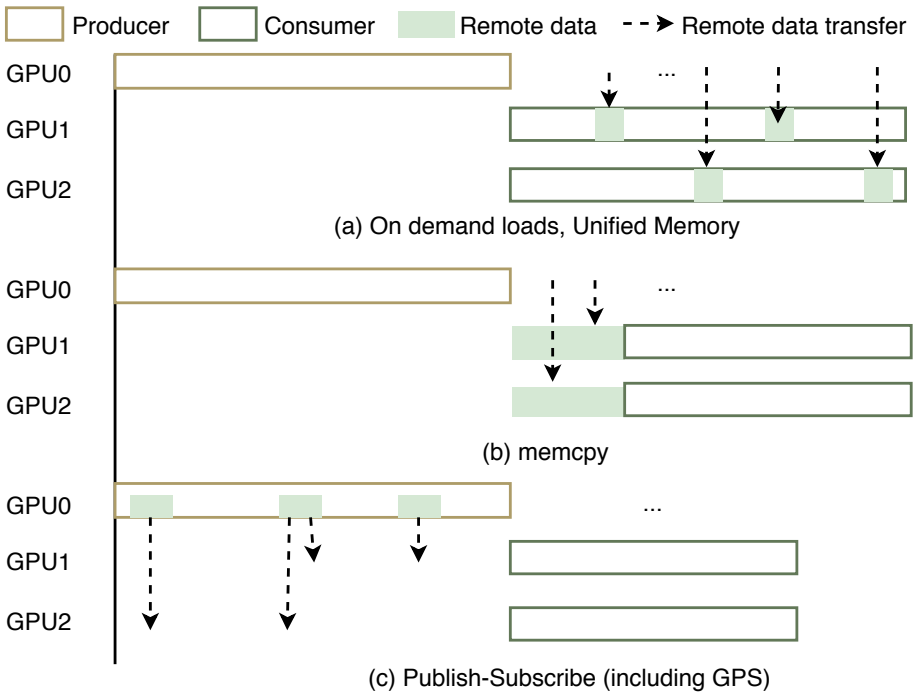


Figure 4.5: Data transfer patterns in different paradigms. In demand/based loads and UM, transfers happen on-demand; in memcpy, they happen bulk-synchronously at the end of producer kernel; in GPS, proactive fine-grained transfers are performed to all subscribers.

4.3.1 Publish-subscribe data transfer patterns

GPS differs from other multi-GPU paradigms in the way it proactively orchestrates stores so that loads can complete locally during compute. Figure 4.5 depicts three different multi-GPU data transfer paradigms. The figure illustrates a simple producer-consumer kernel interaction executing in a 3-GPU system. The producer kernel running on GPU0 generates data and consumer kernels running on GPU1 and GPU2 subsequently read the generated data. We discuss below the nature of the remote data transfer patterns under three multi-GPU programming paradigms.

- Under peer-to-peer loads and UM, the consumer kernels fetch data on demand, as depicted in Figure 4.5(a). As described in Section 4.2.2, these on-demand transfers pay substantial performance penalties.

- Memcpy-based transfers copy data to consumer GPUs at the end of the producer

kernel (in a bulk-synchronous fashion) to make the required data available locally at GPU1 and GPU2 at the start of the consumer kernels, as shown in Figure 4.5(b). These transfers are often able to saturate the interconnect during the inter-kernel transfer phase, however execution typically is stalled while the copies are in progress. As a result, the overall GPU efficiency is lowered because there is no overlap of computation with communication.

- Figure 4.5(c) depicts a typical publish-subscribe use case: GPUs subscribe to the regions they intend to access before kernel execution begins. The hardware supporting the GPS address space then orchestrates the forwarding of proactive remote writes to the targeted pages' respective subscribers as soon as the data is generated by the producer kernel. In this way, GPS-allocated data is available locally on each GPU before the consuming kernel requires it. Therefore, the consuming kernel can access the data at the GPU's full local memory bandwidth, resulting in large performance advantages.

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 next.

4.3.2 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 also contains a set of aggressive coalescing optimizations to ensure that the network is utilized as efficiently as possible. We describe the former here, and the latter in Section 4.3.4.

GPS provides both manual and automatic mechanisms to manage page subscriptions. We describe the general mechanisms below, the APIs in Section 4.4, and the implementation

in Section 4.5.

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. GPS uses subscribed-by-default profiling. Although the early over-subscription initially transfers more data than is required, unsubscribed-by-default profiling incurs stalls due to remote access on first touch similar to the Unified Memory model, and hence is more expensive. To implement the subscribed-by-default policy, our GPS implementation adds architectural support for sharer tracking (see Section 4.5). This information then feeds into the subscription tracking mechanism used to orchestrate the inter-GPU communication.

It is important to note that for both mechanisms the subscriptions are hints to GPS 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. Going to a remote GPU imposes a performance penalty, but does not break functional correctness.

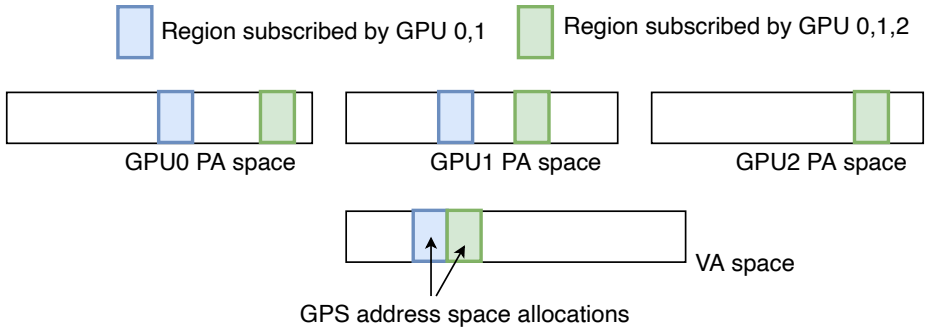


Figure 4.6: GPS address space: Allocations made are replicated in the physical memory of all subscribers

4.3.3 The GPS address space

To enable programmers to incorporate GPS features into their applications, GPS provisions the GPS *address space*, which is an extension of the conventional GPU virtual address space. As shown in Figure 4.6, all allocations made in the GPS address space have local replicas in all subscribing GPUs' physical memories.

Loads and stores are issued to GPS pages in exactly 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.

4.3.4 Aggressive coalescing and functional correctness

While publish-subscribe models have been proposed in the past, a key innovation of GPS is the way it exploits the weakness of the GPU memory consistency model described in Section 4.2.4. In particular, GPS performs aggressive coalescing of stores to GPS pages before those stores are forwarded to other GPUs, as described below.

Coalescing weak writes and `cta`- and `gpu`-scoped strong writes: As described in Section 4.2.4, the GPU memory model only requires weak writes to be made visible to other threads after the next synchronization operation with a scope that includes those threads. 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 get 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 and saving valuable inter-GPU bandwidth. The coalescer must be flushed only upon `sys`-scoped synchronization (e.g., memory fences), including the implicit release operation at the end of every grid.

The GPU memory model also does not require cache coherence, i.e., a consistent store visibility order, unless stores are from the same thread or some form of synchronization is used. 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 also 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.

Strong `cta`-scoped and `gpu`-scoped writes behave similarly: they must be made visible to threads outside of the CTA or GPU, respectively, only after a synchronization operation

with a scope that includes those other threads. Likewise, they can be multicast without concern about memory ordering for the same reason that weak writes can. GPS uses this delayed visibility requirement to aggressively coalesce strong stores with `cta` or `gpu` scope as well.

`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 4.5.3.

The design choices described above ensure that GPS can deliver consistent performance gains without breaking backwards 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.

4.4 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 GPU driver, similar to the existing CUDA API. Listing IV.1 shows sample code.

Memory allocation and release: GPS provides the `cudaMallocGPS()` API call, a drop-in replacement for the baseline CUDA `cudaMalloc()` (for GPU-pinned memory) or `cudaMallocManaged()` (for Unified Memory) APIs, to allocate pages within the GPS address space. This API allocates memory in the GPS virtual address space and backs

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

it with physical memory in at least one GPU. At allocation, the programmer can pass an optional `manual` parameter to indicate 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. Because the driver knows the status of all GPU memory pages, no new API is needed for returning GPS pages to the GPUs' free memory pool.

Manual subscription: To allow expert programmers to explicitly manage subscriptions, GPS overloads the existing `cuMemAdvise()` API for providing hints to UM with two additional hints to perform manual subscription and unsubscription. Specifically, GPS uses the 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 4.3.2, 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 subscribed to a page 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).

4.5 Architectural Support for GPS

We now describe one possible GPS hardware implementation that extends a generic GPU design, such as NVIDIA's recent Volta 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.

4.5.1 GPS memory operations

There are four basic memory operations that GPS must support.

Conventional loads and stores: GPU loads and stores 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 4.7 shows the common case paths taken by writes and reads to GPS pages. 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 4.7. 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 4.5.2) if there is a hit, or it issues remotely to one of the subscribers.

GPS stores: Stores to GPS pages initially proceed as normal stores, as shown by W1 and W2 in Figure 4.7. 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 using this translation (W4), ensuring that subsequent local reads from the same GPU thread will observe the new write, a requirement of the existing GPU memory model. In addition, whether or not there is a local replica, the virtually addressed write is also forwarded to the GPS units for possible replication to remote subscribers (W3, W5, W6).

4.5.2 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 [116]) to indicate whether a given page is a GPS page. When this bit is set, the virtual address is

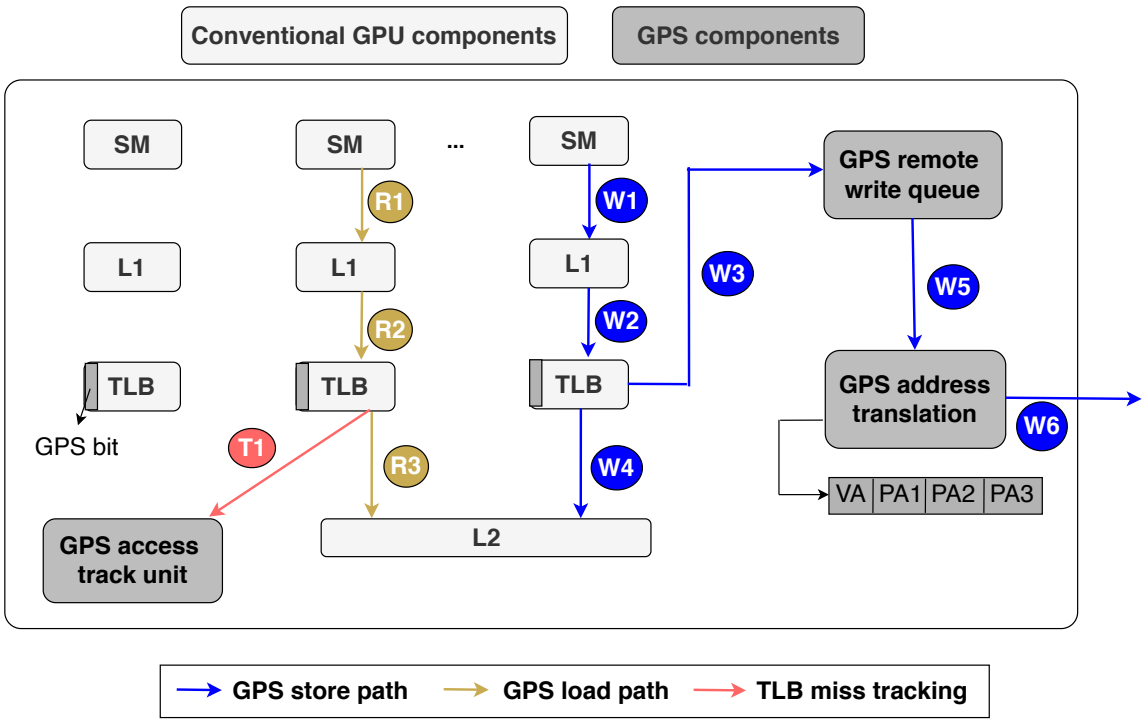


Figure 4.7: Modifications to the GPU hardware needed for GPS provisioning. Alternate design choice explored in Section 4.5.3.

GPS-enabled and 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. Importantly, the GPS page table lies off of 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 being aggressively coalesced, and so the additional latency of the GPS address translation unit can either overlap with the coalescing period, and/or will add just a small additional latency beyond what is already being added. By design, these writes are not required to become visible until the next synchronization boundary anyway, 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 4.7. The GPS-PTE is sized at GPU initialization based on the number of GPUs in the system. With 64KB pages, for a Virtual Page Number (VPN) size of 33 bits and Physical Page Number (PPN) size of 31 bits [116], 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 [127]. As discussed later in Section 4.7.3, 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 of remote writes: If a store's translation information indicates it is to a GPS-enabled address, then the GPS write proceeds to the remote write queue (W3). 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. We set this high watermark to one less than the capacity of the buffer in our evaluation, to maximize coalescing opportunity. On draining, the flushed entry moves to the GPS address translation unit.

If the buffers in the address translation unit are full, it back-pressures execution. We report on stalls due to this back-pressure later in our evaluation. In our default configuration with 64 entries, the GPS-write buffer requires 8.5KB of SRAM storage. The remote write

queue unit must fully drain at synchronization points.

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: To optimize subscription management, GPS 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 4.4, GPS tracks the access patterns during a profiling phase, after which it unsubscribes each GPU from pages they did not access.

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 4.7). As TLB misses are infrequent, yet cover all pages accessed by the GPU, the bandwidth required to maintain this bitmap is low (typically only 1.4 TLB misses per thousand cycles [127]); tracking a 32GB virtual address range, the bitmap requires 64KB of on-chip memory which is smaller than the current GPU L1 cache size. The total area and energy consumed by these hardware extensions are negligible relative to the GPU SoC.

4.5.3 Discussion

Coalescing in the L2 cache: An alternative implementation strategy for GPS is to reuse the GPU's L2 cache to implement the remote write queue, reserving a small number

of L2 cache lines for the GPS write queue rather than provisioning a dedicated structure. The paths taken by conventional and GPS accesses remain the same as in the dedicated write queue design.

Given that the GPS remote write queue amounts to only a few kilobytes of state and the L2 size is megabytes (6MB in Volta GPUs and 40MB in Ampere GPUs), cache capacity impact is negligible. Nevertheless, to isolate the impact of the GPS remote write queue from cache capacity reduction effects, we choose to implement a dedicated remote write queue in our evaluation.

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). From this point on, accesses to the page are all issued to the GPU hosting the single physical copy. This approach ensures coherence and same-address ordering for this access and all future accesses to the page.

As mentioned in Section 4.3.4, `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.

4.6 Experimental Methodology

To evaluate the benefits of GPS, we employ a trace based simulation methodology as described below. We compare its performance against many multi-GPU programming paradigms that exist today.

4.6.1 Simulation setup

We model GPS using a trace-based multi-GPU simulation methodology. We simulate GPS in the context of a system with four NVIDIA Volta-like GPUs connected using PCIe in a tree topology. The design parameters are shown in Table 4.1. We run our application suite on a real multi-GPU system comprised of four NVIDIA V100 GPUs to extract memory access traces for all loads/stores. Our tracing infrastructure imposes only a 1.2% slowdown relative to native execution; as such, the time delta between two trace entries is a good approximation of the computation time between two memory operations. We use these time deltas to properly pace the injection rate of memory transactions into our simulation model while accounting for back-pressure from stalls.

We model the GPS memory system and GPU interconnect as depicted in Figure 4.7 using a detailed interconnect model derived from BookSim [60]. We calibrate the link and switch parameters to match several PCIe generations. When strong scaling is bound by inter-GPU communication, hardware profiling shows the interconnect becomes the primary performance bottleneck. Hence a cycle-accurate interconnect model along with a trace-based GPU model that accounts for back-pressure effects from interconnect bandwidth utilization can capture the key performance impacts of the GPS design.

We evaluate a suite of applications from various domains, ranging from scientific computation to graph algorithms, and include several iterative applications from the Tartan benchmark suite [77]. These applications were chosen as test cases for the GPS concept because they all exhibit poor strong scaling in their native multi-GPU versions on real systems. All our application variants are written in CUDA and compiled using CUDA 10.

GPU Parameters	
Cache block size	128 bytes
Global memory	16 GB
Streaming multiprocessors (SM)	80
CUDA cores/SM	64
Memory bus width	4096-bit
L2 Cache size	6 MB
Warp size	32
Maximum threads per SM	2048
Maximum threads per CTA	1024
GPS Structures	
Remote write queue	64 entries
Remote write queue entry size	135 bytes
TLB	4-way set associative
TLB hit latency	20 cycles
TLB miss latency	2 DRAM accesses
TLB size	32 entries
Virtual address	49 bits
Physical address	47 bits

Table 4.1: Simulation parameters, based on NVIDIA V100 GPUs.

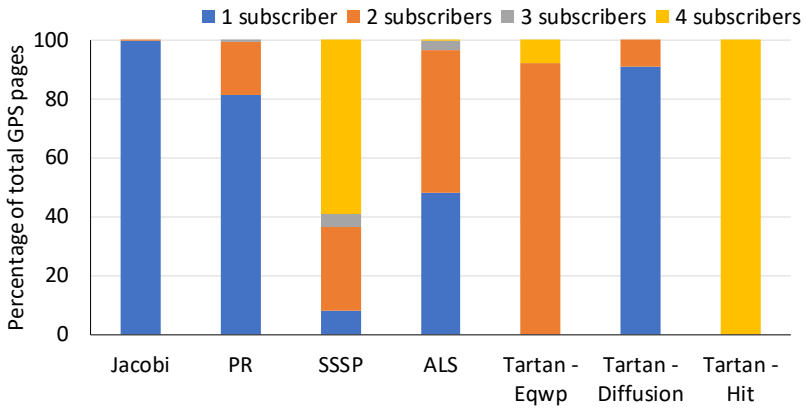
4.6.2 Multi-GPU programming paradigms compared

To demonstrate the ability of GPS to improve multi-GPU scalability, we compare it with several contemporary multi-GPU programming paradigms; namely, unified memory, peer-to-peer loads, and inter-kernel memcpy, as discussed in Section 4.3.1. We simulate UM by modeling a 50 microsecond per-SM stall each time a 64KB page migration must be performed, following the conventions of NVIDIA's UM implementation [175]. We also implement two versions of GPS:

GPS with automatic subscription management: We modify applications to enable GPS and automatic subscription as discussed in Section 4.4. In Section 4.7 we will discuss the effectiveness of the load tracking hardware at reducing over-subscription to minimize inter-GPU traffic.

Explicit broadcast (EB): This model is a variation of the GPS paradigm wherein every store to the GPS address space is issued to all GPUs in the system. It is an overzealous

(a) Percentage of application pages that are read-write and read-only. The read-write pages are allocated as GPS pages.



(b) Subscriber distribution for GPS allocated pages. All pages with less than 4 subscribers result in interconnect bandwidth savings compared to explicit broadcast.

Figure 4.8: Page distribution across the conventional and GPS address spaces.

implementation of GPS, without the ability to unsubscribe from portions of the address space. Whereas EB offers some advantage over the memcpy baseline because it more easily overlaps computation and communication, it often transfers far more data than is necessary.

4.7 Experimental Results

The GPS system relies on fine-grain, 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

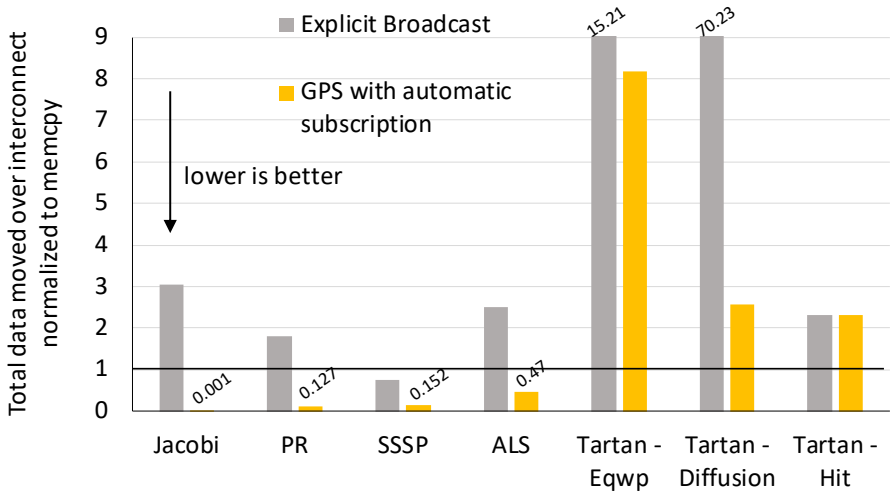


Figure 4.9: Total data moved over interconnect for explicit broadcast and GPS normalized over the amount of data moved during memcpy. Greater than 1 indicates more data sent when compared to memcpy, while less than 1 indicates reduction in data sent over memcpy.

savings. In this section, we first analyze the benefits of the subscription tracking mechanism and then present performance results across different interconnect generations.

4.7.1 Benefits of Subscription Tracking

Figure 4.8a shows the percentage of application pages that are read-write versus read-only in our workload suite. Assuming all read-write pages are initially allocated and subscribed in the GPS space by all four GPUs, using the automatic subscription scheme, Figure 4.8b shows the percentage of GPS pages that have 1–4 subscribers at the end of program execution. Whereas some applications do perform all-to-all transfers for nearly all pages in the GPS space (Tartan - Hit), other applications (like Jacobi) require only one 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 4.9 compares the total data moved over the GPU interconnect for both GPS and

the explicit broadcast implementation, normalized to the inter-kernel memcpy variant of our workloads. Generally, employing explicit broadcast causes large increases in interconnect traffic over manual memcpy because data is often needlessly replicated to GPUs that never consume these updates. So, while explicit broadcast enables the overlap of interconnect transfers with computation, this additional data overwhelms the available bandwidth and degrades performance.

Compared to explicit broadcast, the GPS design drastically reduces the total data transferred over the interconnect via page unsubscription in most cases. However, compared to the memcpy paradigm, there may be improvement or degradation in the amount of data that is transferred across the interconnect. We observe improvements when GPS allows small granularity updates to occur within a page, in contrast to bulk DMA transfers. When degradation occurs, it is typically due to the GPS write combining buffer failing to coalesce multiple writes to the same block effectively, but if they do not saturate the interconnect they will not typically stall GPU execution.

4.7.2 Multi-GPU performance

Figure 4.10 shows the multi-GPU application speedup over a single GPU for the five different programming paradigms described in Section 4.6. In addition to these application variants, we provide a *compute-only* comparison, which is the upper bound in achievable multi-GPU performance if 100% of data were always accessible locally at each GPU (i.e., it ignores all transfer costs).

The unified memory paradigm is not effective for multi-GPU programming, despite having attractive programmability properties, due to the cost of page faults necessary to migrate data between GPUs. While work is ongoing [11, 175], it is unclear if executing page faults on the GPU's critical load path in multi-GPU systems will ever scale well.

Using programmer-directed memcpy to optimize locality is the most common programming technique today. Memcpy performs well for Tartan-Diffusion, but on average does not

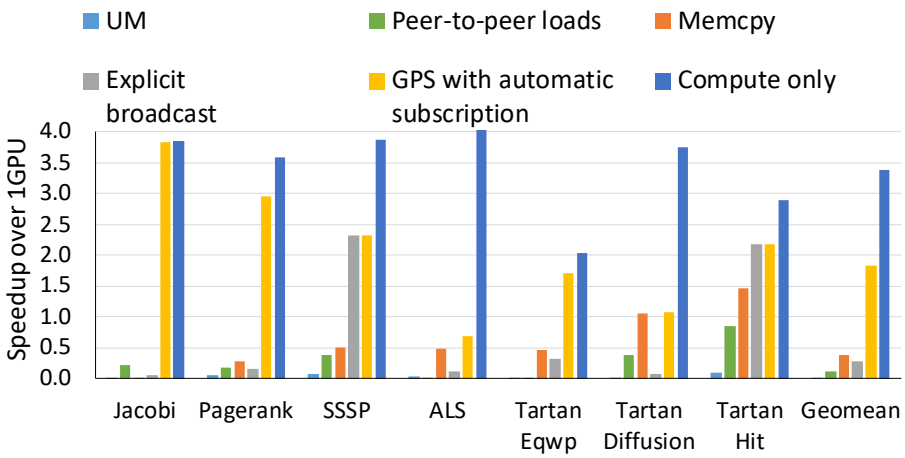


Figure 4.10: 4-GPU speedup of different paradigms

achieve any improvement over a well-optimized single GPU implementation. This finding is significant because it demonstrates that it is not just possible, but likely that multi-GPU parallelization will result in *no strong scaling speedup* using this technique.

Peer-to-peer loads to remote GPUs perform slightly better than unified memory, with similarly simple programming semantics, but again many applications fail to achieve any speedup over the single-GPU implementations. Finally, explicit broadcast suffers from poor performance primarily due to interconnect bottlenecks, demonstrating the significance of subscription management mechanism of GPS.

The GPS methodology ensures a good overlap of communication and compute; at the same time decreasing the unwanted stores to GPUs that do not require a given page. As a result, GPS outperforms the next best paradigm by 3.3×, with an overall average speedup of 2.5× (out of 3.3× possible) over the single-GPU programs.

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 sub-portion of the (64KB) page, yet the interconnect will still transmit all updates to this page. Second, GPUs typically issue interconnect transfers at cache line granularity. We observe that for many HPC applications, not all bytes within a cache block are updated, resulting in the transfer of some unneeded bytes. Third, although the write

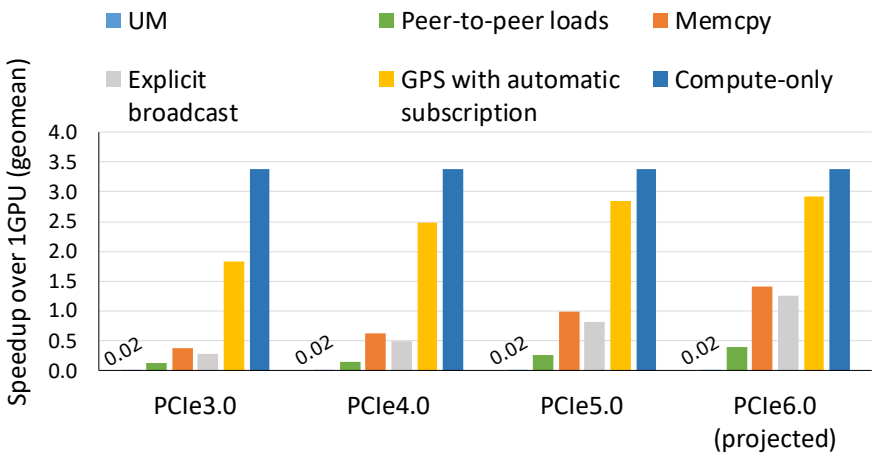


Figure 4.11: Sensitivity to interconnect bandwidth.

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.

Figure 4.11 provides the geometric mean performance of each programming model shown in Figure 4.10, while increasing the throughput of the inter-GPU interconnect. Our results show that despite expected dramatic improvements in PCIe (PCIe 6.0 projected to be up to 128GB/s) and other inter-GPU interconnects [6], good strong scaling is unlikely to be achieved using traditional GPU multi-programming paradigms. Conversely, as GPUs move to higher performance interconnects in the future, GPS will approach the limits of performance scalability by making efficient use of inter-GPU bandwidth.

4.7.3 Quantifying GPS design parameters

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 4.12 studies the sensitivity of buffer size versus achieved performance. With just 64 buffer entries all applications achieve near peak performance. Further performance is difficult to capture due to random writes that have neither temporal nor spatial locality.

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 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 [99] and we initially expected the GPS-TLB to require a similar number of entries. However, Figure 4.13 shows 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.

4.8 Related Work

Prior work [7, 89, 171, 17, 11, 68] has explored the use of various hardware and software mechanisms to improve performance in multi-GPU systems. Our work explores employing the publish-subscribe paradigm for strong scaling in multi-GPU systems. Prior work [150, 129, 130] has also proposed different mechanisms for inter-GPU coherence. GPS does not implement an expensive coherence protocol because it is not a requirement for GPU memory model compatibility.

Several teams have studied prefetching in the context of single GPUs [139, 72, 73] 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* [109] to improve 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

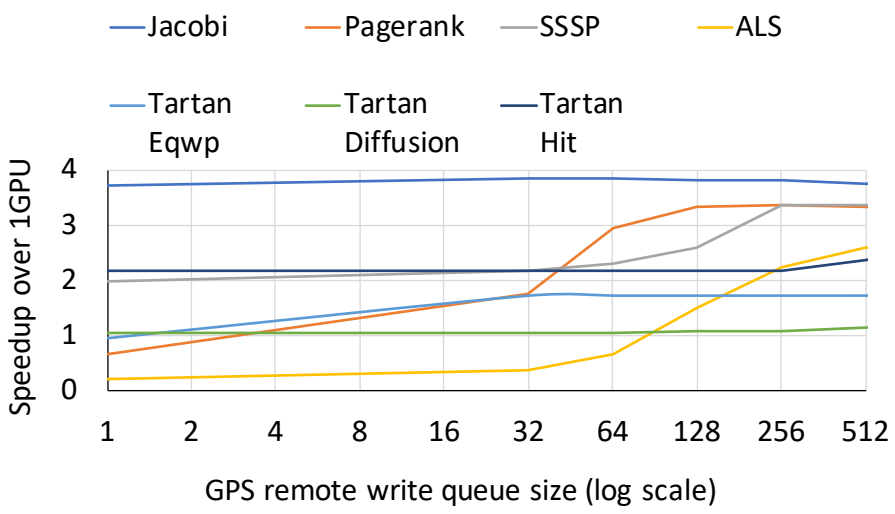


Figure 4.12: Performance sensitivity to GPS write queue size.

GPS-like system.

The publish-subscribe communication paradigm for distributed interaction has been explored by prior work [40, 2, 9, 10]. Hill et al. [50] 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. [32] develop and implement a memory placement algorithm in Linux to address traffic congestion in modern NUMA systems. Many other works [173, 1, 75, 128, 38] perform NUMA-aware optimizations to improve performance and hardware-based peer caching has been explored but is yet to be adopted by GPU vendors [83, 126, 31, 143, 12]. Recently DRAM-caches for multi-node systems [28] have been proposed as a way to achieve large capacity advantages.

Prior work has also explored scoped synchronization for memory models [52, 119, 161, 42, 49, 84]. Non-scoped GPU memory models are simpler [142], but do not permit the same type of coalescing as GPS, which makes explicit use of scopes.

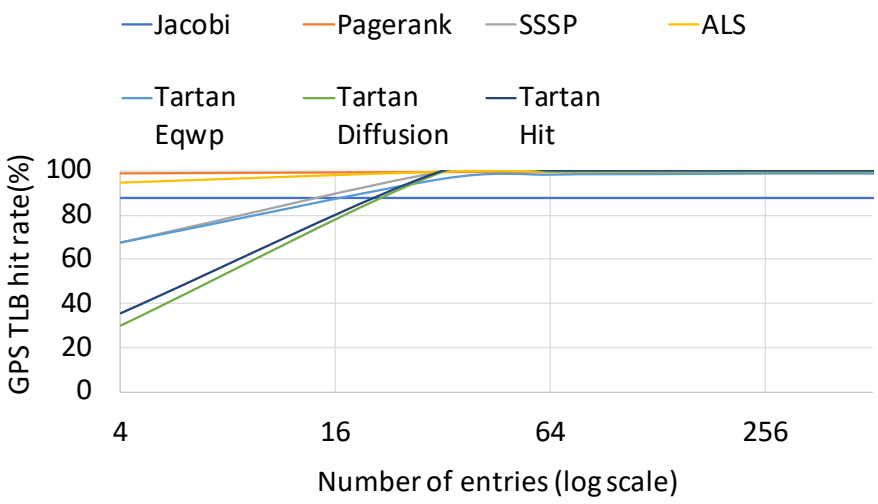


Figure 4.13: GPS TLB hitrate vs. TLB entries.

4.9 Conclusion

In this work we have proposed and evaluated GPS, a set of architectural enhancements 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 all subscribers. This enables the subscribers to read data from local memory at high bandwidth. GPS provides these advantages while retaining compatibility with the conventional GPU programming and memory models. Evaluated on a model of 4 NVIDIA V100 GPUs connected by varying interconnect architectures, GPS offers a speedup of up to 3.8× (2.5× on average) over 1 GPU and performs 3.3× better than the next best available multi-GPU programming paradigm.

Strong scaling in multi-GPU systems is a challenging task. This work has shown that building blocks such as GPS can successfully enable GPU programmers to perform inter-GPU transfers efficiently with minimal programming overhead. We hope this work inspires a new body of research on differentiating the multi-GPU programming paradigm from single GPU programming semantics to help extract the best possible performance out of next generation multi-GPU systems.

CHAPTER V

FinePack: Improving the Efficiency of Fine-Grained Transfers in Multi-GPU Systems*

5.1 Introduction

As demand for compute throughput continues to skyrocket, GPU vendors continue to deliver larger systems comprising more GPUs, more memory bandwidth, and larger interconnection networks. Many problems in the high performance computing and deep learning (DL) domains show excellent weak scaling characteristics and naturally enjoy continued performance scaling as these systems grow. Recently, tasks such as training DL networks with billions of parameters have been scaled to systems with thousands of GPUs thanks to continuous development effort among the GPU computing industry [59, 54].

However, *strong scaling* problems across increasing numbers of GPUs are often accompanied by a super-linear increase in inter-GPU communication. As a result, attempts to strong-scale application performance typically become limited by the inter-GPU interconnect, even at low GPU counts. The strong scaling challenge is exacerbated by the fact that even today's best GPU-to-GPU interconnects deliver bandwidth that is an order of magnitude smaller than the GPU's locally attached memory bandwidth [115, 76]. This inter-GPU bottleneck is so severe that many multi-GPU programs exhibit slowdowns when compared

*This chapter based on [96]

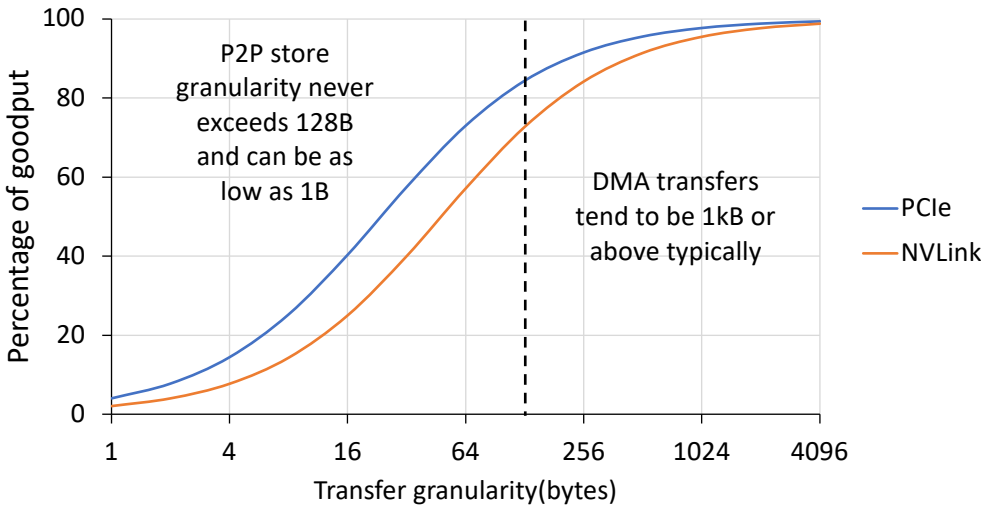


Figure 5.1: Percentage of useful bytes transferred vs. maximium theoretical throughput, when varying the transfer size. Current interconnects are optimized for 128B and larger transfers.

to their single GPU counterparts even after tedious manual optimization [76, 97, 95].

To improve multi-GPU strong scaling performance, multi-GPU programming models and architectural enhancement have been an area of active research and development over the past few years [97, 95, 11, 171]. Recent work has shown the utility of programming models built around proactive peer-to-peer stores: where individual store operations issued by the compute cores on one GPU are routed directly to the memory of a remote GPU [97, 95]. Proactive peer-to-peer stores are efficient, as they do not stall the issuing GPU core, and provide a natural overlap of compute and communication, which is necessary to keep GPU compute units fully occupied. They also maintain a familiar shared-memory programming model and fall within the standard GPU memory consistency model, thus maintaining good programmability when porting applications from one to many GPUs.

However peer-to-peer (P2P) stores result in inefficient small-grain transfers over the GPU-interconnect. We profiled five applications (described in Section 5.4) and found that on average over 75% of inter-GPU transfers initiated by P2P stores carry a payload smaller than 32B, drastically smaller than the typical granularity of DMA operations in GPU programs. Figure 5.1 shows the calculated goodput (number of useful data bytes divided by total

bytes sent) of two common GPU interconnect protocols across a range of transfer sizes. Small P2P issued stores that traverse the inter-GPU interconnect achieve poor interconnect efficiency compared to multi-KB DMA transfers. Due to framing and link-level interconnect protocol overheads, 32B transfers are roughly half as efficient as transfers of 128B or larger. This efficiency loss is so dramatic that even though P2P-store based application design can improve communication and compute overlap, this improvement is typically more than offset by the decrease in interconnect performance, resulting in a net performance loss. Thus, an ideal scalable multi-GPU system must both support a programming model that enables overlap of compute and communication and achieve high interconnect efficiency for both small and large transfers alike.

In this paper, we propose FINEPACK, a set of I/O interconnect and GPU architecture enhancements that improve interconnect efficiency of fine-grained peer-to-peer store operations. By exploiting the GPUs' weak memory model and the application's spatiotemporal locality, we show that it is possible to dramatically improve small transfer efficiency without requiring any changes to the software interface. Fundamentally, FINEPACK temporarily buffers peer-to-peer stores before they are issued over the interconnect, aggregates multiple stores between the same source-destination GPU pair and efficiently compresses and repacketizes the stores using a base+offset compression scheme to minimize framing and link-level protocol overhead. On the receiving side, the write operations are disaggregated before being forwarded to the target GPU's memory system. Both the sender and receiver GPU memory systems remain unmodified; only the interface to the interconnect is changed. Through coalescing and compression, FINEPACK drastically decreases both framing and link-level protocol overheads of small writes and the wasted bandwidth arising from repeated issue of remote stores to the same address.

This work advances state of the art in multi-GPU systems and interconnects with the following contributions:

- We identify the need for efficient small granularity stores and show that existing and

emerging interconnects do not handle them efficiently.

- We propose an extension to the link-level interconnect protocol to significantly improve small store efficiency while remaining within the framework of the existing PCIe protocol. Our approach can also be applied to other interconnect protocols.
- We design the GPU architectural enhancements to make use of FINEPACK while remaining transparent to the existing GPU software interface and memory system.
- Through simulation we show that, on a switched PCIe 4.0 interconnect with 4 GPUs, FINEPACK improves interconnect efficiency by 5.1× while providing an average strong scaling performance of over a single GPU capturing 77 % of the available opportunity.

5.2 Background and Motivation

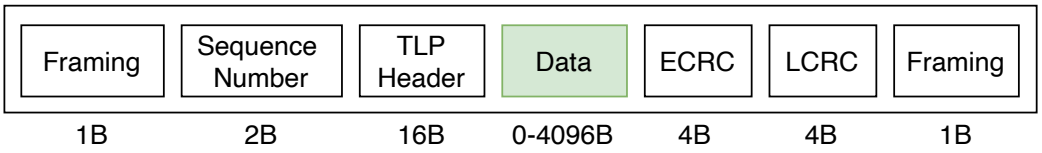
Multi-GPU programming models based on peer-to-peer stores have seen renewed interest due to improvements in PCIe bandwidth and growing adoption of NVIDIA's higher speed NVLink interconnect. Peer-to-peer transfers perform loads/stores directly to the physical memories of the destination GPUs by sending requests and responses across the inter-GPU interconnection network. The GPU's many-threaded architecture has the ability to typically hide the local read memory latency, and thereby to prevent it from causing processor bubbles. However, prior work has shown that because of the lower bandwidth and higher latency of a read from peer GPU memory, performing loads directly from other GPU's memories does not result in good multi-GPU performance [97]. Conversely, the approach of using P2P stores to push data into remote GPUs' memory possesses a natural ability to overlap compute and communication with minimal programmer effort, since the data is naturally propagated to the eventual consumer as it is generated. Since stores are used rather than loads, this communication takes place off of the critical path for most data. When the consumer is ready, the data is already available locally within the target GPU's high bandwidth memory.

Programming models based on P2P stores have emerged because other common multi-GPU programming models all have significant drawbacks. For example, using traditional bulk memory transfers between GPUs using their DMA engines results in non-overlapping phases of compute and communication. While common because they are easy to reason about in the simple case, DMA based transfers that achieve high performance require complex double buffering of data, use of asynchronous concurrent programming streams within the GPU, and can still leave communication time among GPUs exposed without sufficient computation to execute concurrently as the number of GPUs communicating in a system grows. The second common GPU programming methodology, Unified Memory (UM), provides a single memory address space accessible from any processor in the system [48], much like the model when using P2P stores. However, UM implements this abstraction using software page fault-based and/or API hint-based page migration. Prior work has shown that despite many optimizations, the cost of reactively migrating pages to a GPU in an attempt to optimize locality is too inefficient to be utilized in multi-GPU system [95, 1].

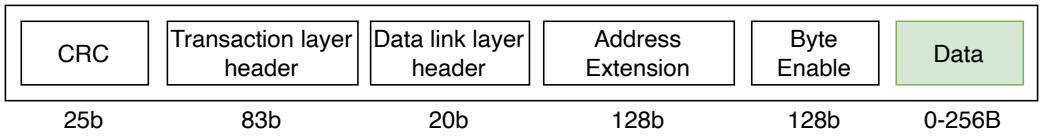
5.2.1 Understanding GPU Interconnect Inefficiencies

If P2P stores are an ideal programming model for multi-GPUs but have not yet been adopted, we must understand what factors are limiting their performance within multi-GPU system designs. Today, modern multi-GPU systems rely on interconnect technologies such as PCI Express (PCIe) [163], NVLink [114], InfiniBand [120], Xe [121], or Infinity Fabric [14] to network GPUs together. Other technologies continue to emerge such as Compute Express Link (CXL) that extends PCIe with support for memory coherency among processors and memory expansions [154]. Because of their early presence in the marketplace, PCIe and NVLink are the predominant interconnects used for multi-GPU systems within a single node today and InifiniBand with RDMA support is the most common multi-node interconnect.

When considering the properties and interaction of programming systems with the protocols used for inter-GPU communication today, we can quantify the sources of interconnect



(a) PCIe 4.0



(b) NVLink 1.0

Figure 5.2: Packet structure of two common GPU interconnects.

bandwidth inefficiency into four categories.

- **Protocol overheads:** Interconnects form network packets by combining the data payload with protocol headers. These headers contain metadata to enable transaction routing and provide different services such as flow control, QoS, reliability, etc. As shown in Figure 5.2, because the header bytes in most packet formats are fixed, the smaller the data payload, the worse the interconnect goodput (fraction of useful data sent over the interconnect), as shown in Figure 5.1.

- **Over-transfer of data:** In situations where the programmer employs bulk DMA-based transfers at the end of compute kernels and these kernels have performed sparse updates to data structures, it is difficult for the programmer to identify the precise subset of memory locations that were updated. Hence, the programmer may end up transferring unwanted data, i.e., data that was not updated in the compute kernel and/or not consumed at the target GPU. Moreover, DMA-based transfer are typically initiated using software APIs which in many cases have to go through many software layers (i.e. runtime, driver, etc.) resulting in additional overheads that become prohibitive if the granularity of the data transfer is too small. Fine grained P2P stores implicitly optimize these overheads by pushing only data that is written using hardware based memory operations. However, they introduce a new problem of redundant transfer.

- **Redundant transfer of data:** When a programmer performs data transfers via weak peer-to-peer stores and these stores exhibit temporal locality, many writes to the data may occur before a final value is produced. The value of the location is only known to be final when the program reaches a synchronization barrier. Until the barrier, sending multiple P2P stores to the same address is redundant and results in wasted interconnect bandwidth. We will later show that this effect is as significant as the protocol overhead when employing P2P writes in GPU systems and must be addressed to achieve high interconnect efficiency.

- **Data padding:** In many interconnects, data is transferred on the wire in units known as *flits*, which represent the amount of data transferred per flow-control unit. When the total payload size is not a multiple of the native flit size the payload is often padded to match the native flit size, resulting in additional wasted bandwidth. In this work, we find this overhead is not one of the primary sources of inefficiency but recognize it may need to be considered when optimizing other protocols with large flit sizes.

5.2.2 The Opportunity for Address Compression & Coalescing

GPU interconnect packets are formed as a result of stores that are destined for memory physically located on a remote GPU. Each store from an individual GPU thread will range from 1–8B. If there is good spatial *and* temporal locality, GPUs will naturally coalesce memory accesses across a *warp* or *wavefront* of 32 threads into a single memory access of up to 128B[†]. However, if no locality arises, either due to the lack of sufficient optimization on the part of the programmer or due to the inherent nature of the problem itself (e.g., in graph or sparse matrix algorithms), no coalescing can be performed, and accesses to remote GPUs are sent over the interconnect with even smaller payloads of 32B or less. Since address spaces today are commonly as large as 48–64b, it takes 6–8B of overhead to transfer a full address along with each packet.

[†]These numbers come from NVIDIA GPUs, but other GPU architectures have similar characteristics.

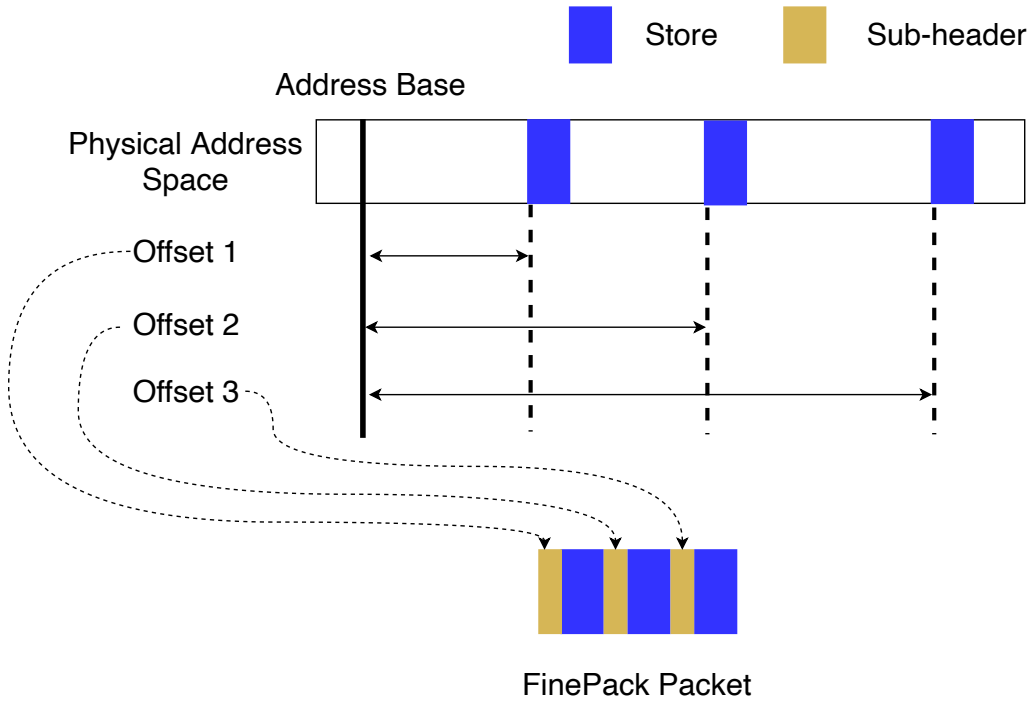


Figure 5.3: Spatial locality enables address compression by partitioning the address into base and offset

This work makes the important observation that by examining the address stream egressing a GPU to other individual GPUs, there is significant opportunity for both compressing packet headers and addresses and reducing redundant writes before packets are sent out over the interconnect. Figure 5.3 illustrates the basic premise on which FINEPACK is based. When examining a stream of writes (egressing a single GPU that are destined for a single remote GPU) we observe that these writes will not use the entire physical address space present in the system because most applications will benefit from some amount of spatial locality. This allows us combine multiple small writes (to different addresses) that have redundant common address bits into a new base plus offset addressing scheme that compresses wasted bits, in addition to saving on the fixed packet overheads shown in Figure 5.2. Further, should any of these writes be destined for the same address (or address range), the GPU's weak memory model allows even more efficient compression/coalescing by simply overwriting the data locally before sending the packet out on the interconnect. Exploiting

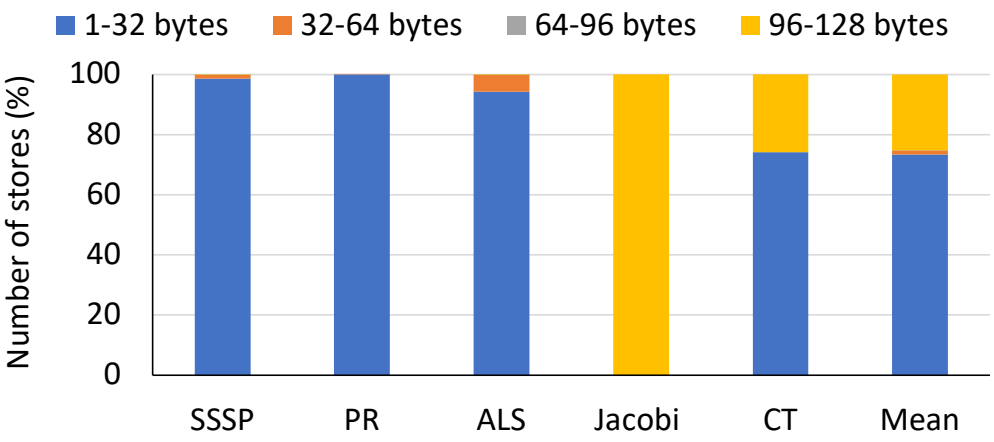


Figure 5.4: Average size of remote stores exiting the L1 cache for the P2P store paradigm.

these two properties with modest architectural modifications can allow fine grain P2P stores to achieve network efficiency that rivals or exceeds that of bulk DMA memory copies and overcome the primary limitations for adopting P2P stores as the preferred method for achieving strong multi-GPU scaling.

5.3 FINEPACK Design

Figure 5.4 shows the fraction of inter-GPU transfers egressing the GPU's L1 caches, broken down by size of transfer. Despite intra-GPU coalescing of small writes occurring within both the SM and L1 cache, many applications not optimized for bulk DMA transfer often emit sub-128B communication. The goal of FINEPACK is to transfer fine-grained stores across the interconnect more efficiently. It does this by compressing multiple individual transfers into a single large transaction. As such, the FINEPACK design consists of two primary components: extensions to the interconnect transaction layer to add a compressed store operation and the microarchitecture that coalesces transactions in a network-efficient manner. We describe these two components in turn below.

5.3.1 FINEPACK Packet Structure

Figure 5.5 shows the FINEPACK packet structure embedded within a PCIe packet. It is based on our insight that for stores belonging to the same source-destination pair, most of the interconnect's transaction-layer fields are (or can be made) identical without changing the stores' semantics. Thus, with a suitable mechanism to perform aggregation at the sender and disaggregation at the receiver, the common header fields of many stores need to be transmitted only once, thus saving precious interconnect bandwidth. In effect, FINEPACK allows stores to share framing and transaction layer header fields across otherwise independent stores.

The existing transaction layer protocol (TLP) header fields retain their original meanings and are largely unchanged. However, the payload of the new transaction type now comprises multiple sub-packets concatenated within a single transaction for better interconnect efficiency. Each sub-transaction contains a new header that encodes specifics of the sub-packets that differ from the outer transaction, notably, a compressed address and size of the sub-packets payload for the individual store.

Table 5.1 enumerates the fields in the outer PCIe TLP packet[‡]. Among the required fields of the PCIe TLP, all fields except the address, data length, and byte enables can be made common for stores with the same source and destination. We repurpose an unused encoding in the type field to indicate the new FINEPACK transaction type. The address field in the outer TLP packet signifies a base address for all FINEPACK sub-packets. The length field now represents the cumulative length of the FINEPACK sub-packets (and can be used, e.g., for buffer management). The first byte enable field is unused in a FINEPACK transaction; however, since the PCIe length field is 4B-aligned, the last byte enable field is used to specify the two least-significant bits of the total FINEPACK payload size.

Each FINEPACK sub-transaction header consists of an address component and data length component. The address field represents an offset that is added to the base address in

[‡]We use PCIe as an example, but other interconnect transaction layers, such as NVLink, are similar.

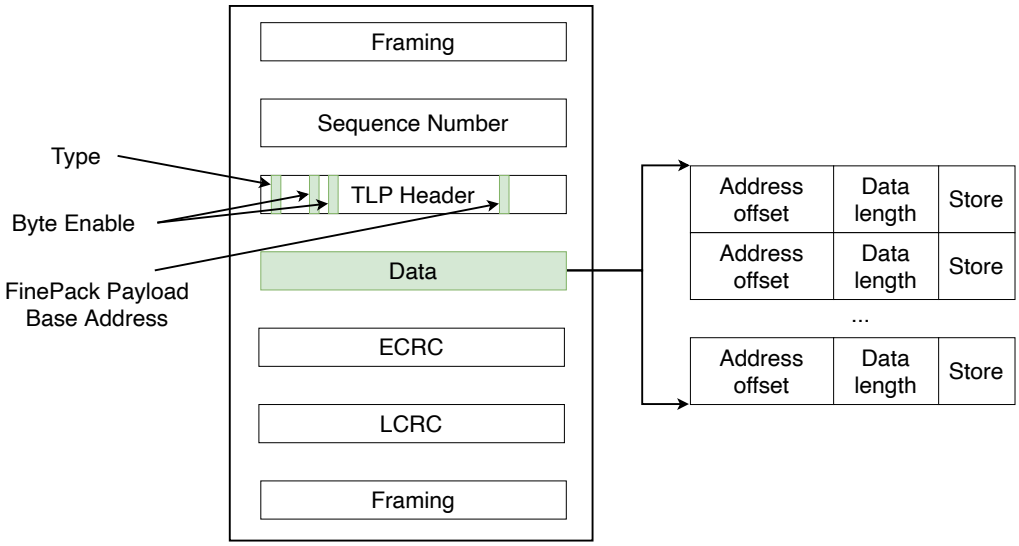


Figure 5.5: FINEPACK packet structure embedded within PCIe.

Bits	Field	Use in FINEPACK outer packet
10	Length	Total length of FINEPACK payload
2	Attribute	Standard PCIe meaning
1	Error/Poisoned	Standard PCIe meaning
1	TLP Digest	Standard PCIe meaning
3	Traffic Class	Standard PCIe meaning
5	Type	**Encoding indicating FINEPACK**
2	Format	Standard PCIe meaning
4	First BE	**0 (Not needed by FINEPACK)**
4	Last BE	**Set relative to FINEPACK payload**
8	Tag	Standard PCIe meaning
16	Requester ID	Standard PCIe meaning
62	Address	**FINEPACK payload base address**
—	Data	**FINEPACK payload**

Table 5.1: PCIe transaction layer protocol (TLP) header fields, as interpreted for FINEPACK packets. Most fields retain their standard meaning.

the outer TLP packet header. The length field describes the length of the inner transaction payload. Although the PCIe header address and length fields are 4B-aligned, the FINEPACK inner transaction format uses 1B-aligned address and length fields to more flexibly support small writes. The four bits of sub-transaction header needed to capture 1B alignment are denser than the eight bits required to propagate the first and last byte enable fields in each sub-transaction header.

The number of bits reserved for addresses and lengths in the sub-transaction header

format is a design parameter that can be adjusted for different manifestations of FINEPACK. For the evaluation in this paper, we sweep the header size across different byte counts as shown in Table 5.2. In each case, ten bits are reserved for the length field, and the remaining bits are used as address offsets. As our results later confirm, 4–5 bytes for the sub-transaction offset (corresponding to 4MB or 1GB of addressable range per sub-packet, respectively) is often the sweet spot between having reasonable overhead per sub-transaction header but still allowing significant addressable range within the boundaries of the outer transaction. FINEPACK augmented PCIe implementations consumes buffers and credits in the same way as a variable length memory write transaction are currently specified on PCIe.

5.3.2 FINEPACK Architectural Components

We now describe one possible FINEPACK hardware implementation that extends a generic multi-GPU design. The overall architecture is shown in Figure 5.6. Our hardware proposal comprises three major components. First is a *remote write queue* that collects outgoing GPU stores at the GPU network egress port and buffers them in an attempt to combine them into larger transactions. Second, there is a *packetizer* to re-packetize the combined stores before sending them to the network using the extended TLP packet format described in Section 5.3.1. Third, there is a *de-packetizer* at the destination GPU's ingress network port to break the aggregated transaction into individual stores before issuing them to the target memory. But PCIe componentry in the system such as switches or the CPU's root complex (if transactions are routed through it) can remain unmodified. We describe each FINEPACK component in turn below.

Remote Write Queue: The remote write queue is a dedicated SRAM structure logically attached to the egress port of the GPU. The remote write queue collects outbound store transactions destined for each GPU so that they can be coalesced into a single outer FINEPACK transaction. Store transactions with addresses that fall within the address window, determined by the address base and the range of address offset, are coalesced into FINEPACK

	Sub-transaction header bytes				
	2	3	4	5	6
Length field bits	10	10	10	10	10
Address field bits	6	14	22	30	38
Addressable range	64B	16KB	4MB	1GB	256GB

Table 5.2: The number of bytes used for sub-transaction headers is a tradeoff between more overhead per sub-transaction header and more addressable range per outer transaction.

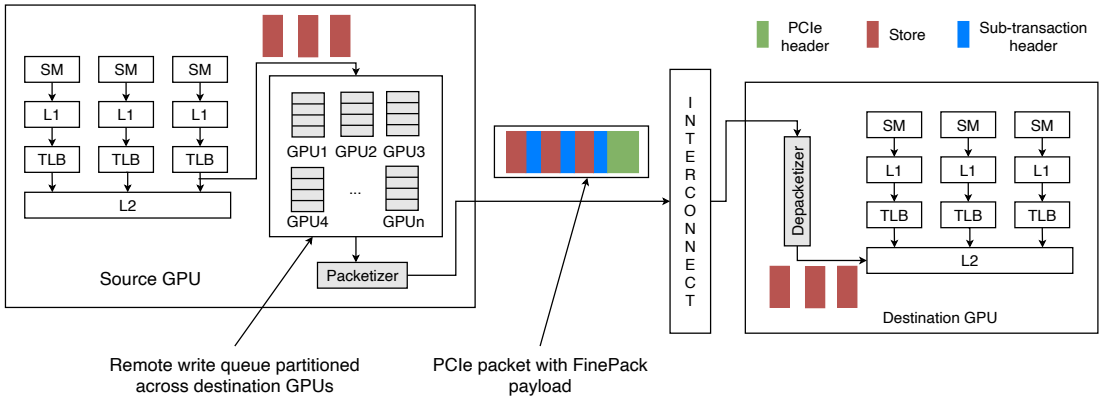


Figure 5.6: FINEPACK architecture.

packets as long as the maximum remote queue size and the maximum PCIe payload size are not exceeded. The queue is logically partitioned across the total number of destination GPUs in the system so that packets targeting each GPU can be coalesced independently. Within each partition, the SRAM is organized as a fully-associative structure indexed by memory address at 128B granularity. Each entry holds an address tag, 128B of data, and a byte-enable bit for each byte.

An available payload length register is initialized to the maximum payload size and decremented for every store that gets added to the remote write queue. It keeps track of the payload length of the packet to ensure it doesn't exceed the length supported by the interconnect. The first store to arrive at an empty remote write queue will allocate an entry, filling in the address, data, and byte-enable bits. It will also set a base address register associated with the targeted GPU to a value matching the upper $ADDRLEN - OFFSETLEN$ bits of the addresses of other valid entries, where $ADDRLEN$ is the number of physical address bits in the system, and $OFFSETLEN$ is the number of bits in the offset field of the sub-transaction

header (Table 5.2). Each subsequent store will then scan the associative structure to find any matches, according to the following procedure:

1. If the incoming store finds an exact address match, it will overwrite any valid data bits in the matching entry, and it will also update the byte-enables and the available payload length register accordingly.

2. If the incoming store address does not match any existing entries, then it will check the upper bits of its address against the base address register. If there is a mismatch, then the incoming packet cannot be coalesced into the same FINEPACK transaction as the currently queued packets due to insufficient sub-transaction header offset bits. When this happens, the current contents of the write queue are flushed and passed to the packetizer, and the incoming packet becomes the first store of a new FINEPACK transaction.

3. If the incoming address matches the contents of the base address register, but does not match any existing entry, then the incoming packet can create a new entry in the write queue, assuming there is room and decrement the available payload length register by the sub-transaction header plus the store size. If there is no room, then once again, the contents of the queue are flushed to the packetizer, and the incoming store initiates coalescing for a new FINEPACK transaction.

To ensure that the remote write queues buffer sufficient data to be able to target reaching the PCIe maximum payload size of 4kB, the remote write queue is sized to 64 entries of 128B each and 24kB total on a 4-GPU system (not counting tags or byte enables). This size can be easily accommodated on a GPU where each L1 cache is already hundreds of KB.

The remote write queue must be flushed upon receiving a system-scoped release operation, such as a memory fence or the end of a kernel's execution. In GPU memory consistency model terminology, system scope refers to synchronization applied to all devices in the system, and release operations refer to the synchronization performed by a producer when

synchronizing its data with some other consumer. Loosely speaking, a GPU release operation generally requires the implementation to flush in-flight stores to the point at which they become visible to all threads in the specified scope. As such, flushing the remote write queue upon receiving a system-scoped release operation ensures that the FINEPACK scheme remains compatible with the GPU memory consistency model.

To ensure same-address load-store ordering is maintained, an load operation to remote memory with an address that matches a write in the packetizer must flush any matching stores currently queued in the remote write queue. These stores can be flushed individually, or load hits can trigger a flush of the remote write queue just as a synchronization operation would. We did not evaluate this tradeoff further as, by design, our workloads use only remote stores.

Packetizer: The packetizer receives a series of 128B byte-enabled remote write queue entries and converts them into sub-packets in the FINEPACK format described in Section 5.3.1. Each individual remote write queue entry may need to be converted into multiple sub-packets if the enabled bytes are not contiguous, as the sub-transaction header does not contain byte-enables. This set of sub-packets is then concatenated into the data payload of the outer transaction and transferred to the interconnect.

In rare cases in which the pattern of writes produces sparse byte enables, the number of sub-packets required to drain the write queue may exceed the outer packet payload capacity (4KB on PCIe). In such cases, the packetizer splits the sub-packets across multiple outer packets and transfers them independently. Such scenarios are possible, but did not arise in our evaluations.

De-packetizer: When the FINEPACK packet reaches the destination GPU, the de-packetizer de-aggregates the stores into individual memory transactions before sending them to the target memory system. The de-packetizer simply modifies the address field of each de-aggregated packet to add the offset from the sub-transaction header to the base address from the main header. No other changes are needed to the GPU architecture.

5.3.3 Discussion

Effect of small accesses on local memory bandwidth: We note that although FINEPACK ensures reduced packetization overheads on the interconnect, the communication to and from the GPU's local memory still occurs as fine-grained transfers. In general, the GPU's last-level cache and HBM/DRAM have enough bandwidth to match or exceed the rate at which stores can arrive from the interconnect. As such, the existence of fine-grained transfers arriving at the target GPU's local memory system does not further bottleneck the FINEPACK system architecture.

Timeouts: The GPU's weak memory consistency model does not require stores to become visible to remote GPUs until system-scoped synchronization is performed [84]. Therefore, it is functionally correct for the remote write queue to buffer stores indefinitely until a system-scoped synchronization is observed. It is also functionally correct to drain the remote write queue at any time, for any reason. More aggressive flushing, e.g., after some kind of inactivity timeout, will lower the end-to-end latency of store transmission, and can avoid burstiness if the remote write queue is flushed with more stores buffered at once. On the other hand, less-aggressive flushing will keep the coalescing window open for longer, potentially leading to even better efficiency for packets sent over the network. As any choice is functionally correct, the choice of when to drain the remote write queue is therefore simply a knob that can be tuned as needed to achieve higher performance.

Applicability to read accesses: FINEPACK focuses on improving interconnect efficiency for applications that adopt a peer-to-peer store programming model for communication. In such applications, proactive data transfers via stores ensures that subsequent loads complete locally without having to traverse the interconnect on the critical path. Since on-demand loads stall the GPU threads issuing them, performing FINEPACK optimizations for loads will exacerbate their latencies and can further harm application performance. Hence, FINEPACK performs aggressive packing and coalescing only for store operations.

Base Addresses: As described in Section 5.3.2, FINEPACK sets the base address of the

remote write queue to a copy of the address of the first incoming store with the low-order bits corresponding to the FINEPACK sub-transaction addressable range (Table 5.2) masked off.

There are multiple ways to solve the problem of spanning alignment regions. One approach is to use a more sophisticated remote write queue design that dynamically calculates the address range spanned by all currently queued stores, and then flushes only when an incoming store causes the range to exceed the FINEPACK sub-packet addressable range. An alternative design might maintain multiple open outer transactions for each target GPU so that accesses to data structures spanning two aligned regions do not thrash the remote write queue. We evaluated the simple design to avoid unnecessary complexity and because the issues described here did not arise as first-order concerns in the applications we evaluated.

Logical Partitioning of the Remote Write Queue: The amount of SRAM allocated for remote write queue can be partitioned in multiple ways. In our evaluation, we sized the buffer to hold 4KB (the maximum PCIe payload size) for each remote GPU. However, the size could be adjusted as performance needs dictate. If in larger systems it becomes too expensive to allocate that much storage, FINEPACK will scale down gracefully as the number of remote write queue entries shrinks. As described above, it is also possible to allocate more than one buffer partition per remote GPU to avoid thrashing, at the cost of fewer entries per any individual partition. More sophisticated designs might construct the SRAM with full dynamic allocation, rather than partitioning capacity in advance. We leave further exploration of such fine-tuning possibilities for future work.

Compatibility with Memory Ordering Rules: Although the FINEPACK write queue reorders store operations, this does not violate any GPU memory ordering rules. The GPU's weak memory model permits accesses to non-overlapping addresses to be reordered freely (assuming there is no synchronization operation in between). Stores to the same address will simply overwrite each other in the remote write queue, which is also permitted. Standard PCIe ordering rules require store TLPs to remain in order, and this invariant is maintained

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
FINEPACK Structures	
Remote write queue	192 entries
Remote write queue entry size	132 bytes
Base address register	8 bytes
PCIe maximum packet size	4096 bytes
FINEPACK subheader size	5 bytes
FINEPACK subheader address offset	30 bits

Table 5.3: Simulation parameters, based on NVIDIA V100 GPU.

once a FINEPACK transaction is issued, but before and after packetization the reordering is freely permitted on the GPU.

Applicability Beyond PCIe: Although we focus on PCIe for our evaluation, the techniques we have described generalize to other protocols such as NVLink or CXL. Each interconnect technology differs in the details of packet formats and encodings. For example, protocols such as NVLink that use byte enable fields for the entire payload may end up using slightly different encodings of the FINEPACK payload in the outer transaction. Nevertheless, the general approach of compressing multiple small stores into a single payload within a larger outer transaction still applies.

5.4 Experimental Methodology

We evaluate FINEPACK by extending the NVIDIA Architecture Simulator, NVAS [156] comprising a system of NVIDIA GV100 GPUs connected over existing and projected PCIe generations with bandwidths ranging from 32GB/s for PCIe 4.0 to 128GB/s for PCIe 6.0. NVAS is a trace- and execution-driven multi-GPU simulator that exhibits good simulation

fidelity without sacrificing speed and has been correlated across a wide range of benchmarks. We collect application traces at the GPU assembly level using NVBit [157] and replay these traces in the simulator. These traces contain CUDA API events, GPU kernel instructions, and memory accesses. The simulator models timing when replaying traces. Our FINEPACK implementation in NVAS is configured with parameters listed in Table 5.3.

We evaluate a suite of multi-GPU applications described below based on prior work [97].

Single Source Shortest Path (SSSP): We use the Bellman-Ford algorithm to compute the shortest path, i.e., the path between a given vertex and all the other vertices of a graph such that the sum of the weights of the edges in the path is the least [66]. This algorithm is extensively used to compute the minimum distance between different geographical locations, such as road network [53]. Our implementation formulates the computation as a matrix operation similar to prior works [97, 66]. We compute SSSP on the HV15R dataset [35].

Page Rank: Page Rank is used by Google to rank web pages, it works based on the assumption that the more important a web page is, the more likely it is to receive incoming links from other web pages [166]. Our application assigns pagerank to the Wikipedia dataset [35].

Alternating Least Squares (ALS): ALS is an iterative matrix factorization algorithm used to solve collaborative filtering problems. It factorizes a given matrix R into two factors U and V such that $U^T V = R$. Every iteration, the algorithm optimizes one of the two factors while keeping the other factor a constant. The roles are reversed for the next iteration [151]. We perform ALS for HV15R dataset [35] using Stochastic Gradient Descent.

MBIR X-ray CT: Model Based Iterative Reconstruction (MBIR) is a minimization algorithm used in low-dose X-ray CT reconstruction. MBIR can be formulated to admit considerable parallelism over image voxels or detector values, in the sinogram making it well-suited for multi-GPU acceleration. The algorithm operates iteratively in two barrier-synchronized phases wherein data structures are partitioned across GPUs and each GPU requires input data from the previous phase from all other GPUs [81, 87, 134]. Our

implementation seeks to accelerate MBIR via multi-GPU execution by addressing the high remote access cost that arises with GPU scaling.

Jacobi Solver: Jacobi solver is an iterative algorithm, used in numerical linear algebra for determining the solutions of a system of linear equations that are strictly diagonally dominant. It iteratively solves a system of linear equations and computes the solution vector. [165]. Our implementation computes the solution vector for banded matrices, which arise widely in finite element analysis [164, 55].

For each multi-GPU application we have two equivalent implementations that maintain the same data sharing and work sharing characteristics across GPUs, but use a different programming paradigm. The two paradigms are:

Memcpy: This paradigm maintains duplicate copies of data structures among all GPUs to ensure locality at the time of compute. The updates performed by a compute kernel are broadcast after the end of the producer kernel and before the start of the consumer kernels using via cudaMemcpy() API. In this paradigm there are no remote accesses during kernel execution, however data is transferred at coarse granularity preventing the overlap between data transfers and compute.

Peer-to-peer stores: This paradigm employs peer-to-peer stores to transfer data proactively as soon as the data is generated so that the data is resident locally in the consumer GPUs at the time of compute. This paradigm ensures fine-grained overlap of compute and communication without imposing significant burden on the programmer.

FINEPACK works transparently optimizing the peer-to-peer store paradigm providing more efficient bandwidth utilization. To establish an upper bound for the potential benefit of FINEPACK, we also compare it with a system with infinite bandwidth. We modelled such a theoretical system by eliding data transfer time when applications execute using the memcpy paradigm.

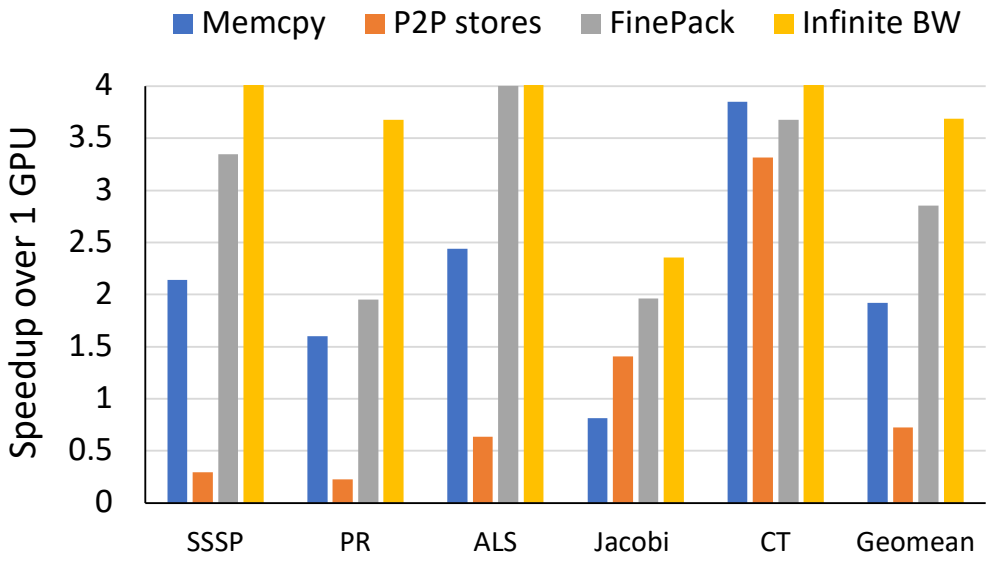


Figure 5.7: 4-GPU speedups

5.5 Results

We next look at the properties and performance of FINEPACK on a system of 4 GPUs connected via PCIe, as described in Section 5.4.

5.5.1 Performance Results

Figure 5.7 shows the speedup of various communication paradigms on a four GPU system normalized to the performance of a single GPU baseline. The infinite bandwidth paradigm shows the total opportunity available upon optimizing inter-GPU communication and on average it is 3.6×, indicating the highly parallel nature of these workloads. The benefits of FINEPACK arise from: (1) its ability to rely on fine-grained peer-to-peer stores that provide natural overlap of compute with communication, (2) its ability to minimize the protocol packetization overhead by aggressively packing multiple stores within the same packet, and (3) its ability to coalesce writes before sending them to the destination GPU thereby reducing the number of redundant writes to the same location.

Figure 5.8 provides a breakdown of the underlying efficiency of the three studied inter-

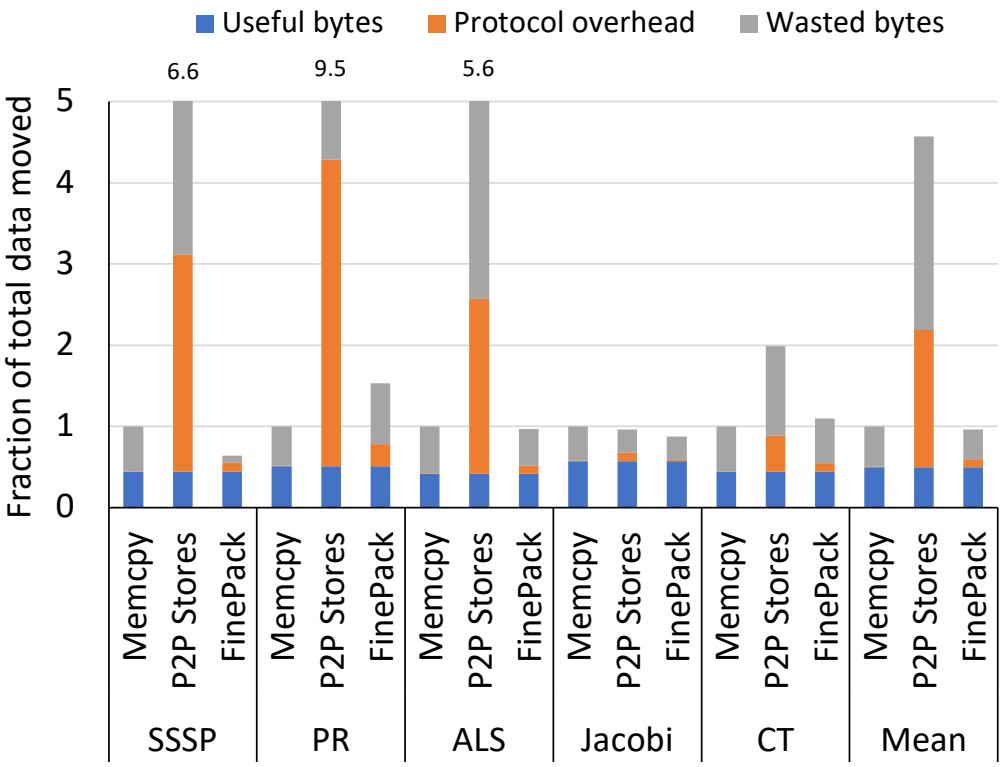


Figure 5.8: Breakdown of total bytes transferred over the interconnect, normalized to inter-GPU memcopy. For each inter-GPU transfer paradigm, these bytes are then categorized into 'Useful bytes', which are read by the destination GPU, 'Protocol overhead' which is the number of bytes required to perform all transfers, and 'Wasted bytes', which is bytes transferred that are never read or are overwritten by the source GPU, before being read by the destination GPU.

GPU communication schemes. The figure shows the total number of bytes transferred over the interconnect in each scheme, normalized to the total data transferred by the memcpy scheme. 'Useful bytes' represents the fraction of total data transferred that is actually read by the destination GPU during its computation; 'Protocol overhead' represents the fraction of interconnect bandwidth spent on framing and header overhead, and 'Wasted bytes' represents data bytes that are either overwritten by later stores to the same location or never read by the destination GPU.

As Figure 5.8 shows, the memcpy paradigm results in negligible protocol overheads due to the coarse granularity of transfers. However, the memcpy paradigm suffers performance degradation compared to FINEPACK due to (1) its inability to overlap compute and

communication, and (2) the high percentage of wasted bytes being transferred due to the coarse-grained nature of these transfers copying data that was not actually updated by the source and/or not actually accessed by the destination.

P2P stores achieve a good overlap of compute and communication, but also exhibit poor performance due to the protocol packetization overheads of the fine-grained transfers as well their inability to exploit temporal locality. P2P stores also suffer from wasted bytes due to the transfers of multiple stores to the same address. These factors result in the total data transferred often being an order of magnitude higher than the amount of data transferred by the memcpy scheme. FINEPACK on the other hand, eliminates the overheads of peer-to-peer stores by packing multiple stores within the same packet, thus decreasing the packetization overhead. It also exploits the relaxed nature of the memory consistency model to aggressively coalesce stores with temporal locality, thus reducing the number of redundant bytes.

5.5.2 Performance Characterization and Sensitivity Studies

Sub-transactions per FINEPACK Packet: Figure 5.9 shows the average number of stores that are coalesced in the FINEPACK packetization buffer before egressing the source GPU. We observe that FINEPACK is able to pack close to 40 stores on average into a single transaction before sending it to the interconnect, thus leading to the protocol overhead reduction shown in Figure 5.8. CT is able to pack fewer stores on average because the individual stores exhibit less spatial locality.

Understanding Address Splitting Decisions: One of the important design decisions that has to be made when considering an architecture like FinePack is how to sub-divide the sub-traction address structure into the address base and offset fields. As discussed in Section 5.3, the number of bytes in the sub-transaction header address field presents a tradeoff: as more address field bytes are used, more stores can be compressed into a single large transaction, but the overhead of each sub-transaction also grows accordingly. As such,

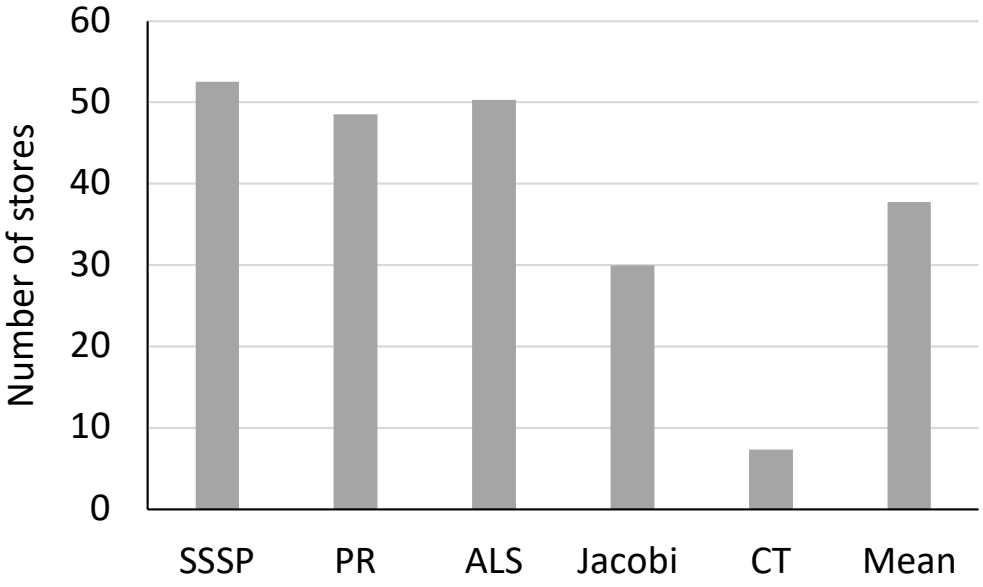


Figure 5.9: Average number of stores aggregated in a single packet by FINEPACK.

we evaluated multiple sub-transaction header sizes to find the sweet spot.

Figure 5.10 shows that performance increases to maximum at 4 sub-transaction header bytes with virtually no change at 5 bytes. Four bytes of sub-transaction header corresponds to 22 bits of address space per outer transaction, meaning that stores within a 4MB-aligned region can be coalesced into a single packet. For our applications, this granularity is sufficient to capture the communication patterns of the workloads evaluated. Beyond 5 bytes, performance begins to decrease because the protocol overhead grows, yet additional stores cannot be accommodated in a single packet. While there is some variation among applications, we observe that for these applications from a variety of GPU computation areas, 4-5 sub-transaction header bytes works well.

Six bytes of sub-transaction header enables addressing of 256GB of memory, which grossly exceeds the total memory capacity of each individual GPU today. Even as the GPU's memory capacity increases in the future it seems unlikely FINEPACK would need to move to 6 byte sub-transaction headers. This is because FINEPACK relies on application locality to minimize the number of overhead bytes in each sub-transaction, not just the maximum size of each GPU's physical memory. This is why 4 and 5 byte sub-transaction headers appear to

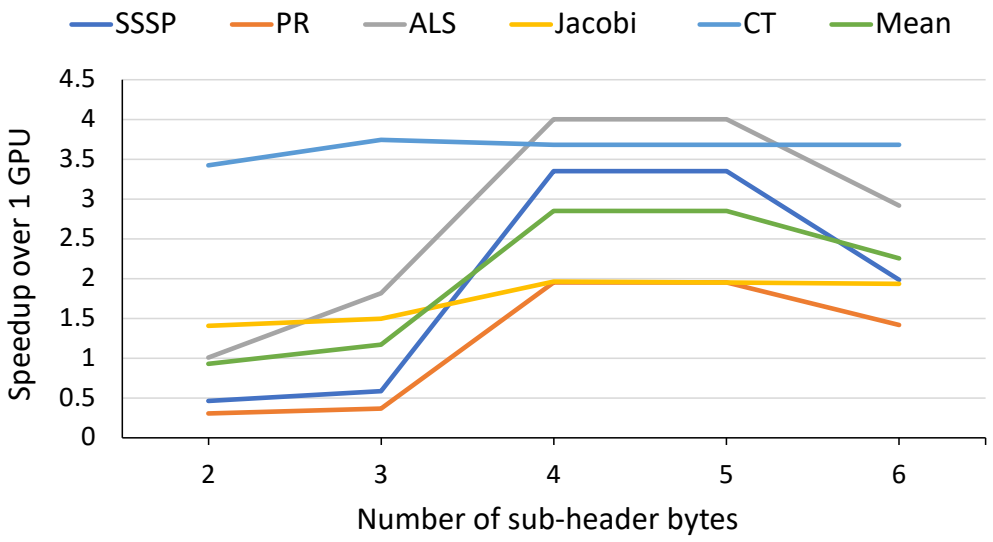


Figure 5.10: FINEPACK performance sensitivity to variation in the number of sub-header bytes.

perform virtually identically, both are large enough to capture the memory stream locality present in the workloads.

Sensitivity to Interconnect Bandwidth: Figure 5.11 shows the geomean performance gains of FINEPACK over memcpy and direct P2P stores when increasing the inter-GPU bandwidth while leaving GPU compute capabilities fixed. We observe that memcpy and P2P stores improve performance significantly with each step in bandwidth. However, at no step (until bandwidth is unlimited) do they reach the performance of FINEPACK because of FINEPACK's ability to better utilize the interconnect by providing improved compute-communication overlap as well as decreasing the protocol overheads.

We expect the interconnect BW to be magnitudes lower than the local memory bandwidth even in future multi-GPU systems. Thus architectures such as FINEPACK which require no GPU programmer intervention are a promising way to exploit address locality in limited physical memories to improve small store performance.

5.5.3 Discussion

Comparison with pother proactive transfer systems: Prior works [97, 95] also employ proactive transfers to improve multi-GPU strong scaling. PROACT [97] is a joint compile and runtime system to achieve intelligent orchestration of shared data in multi-GPU systems. It transparently fine-tunes inter-GPU data movement for the application's needs, thus achieving the interconnect efficiency of bulk transfers while maintaining the programming simplicity of peer-to-peer stores. However PROACT requires per-application profiling and program re-writing to integrate its framework into each application. FINEPACK on the other hand, is completely programmer transparent and offers performance gains without the need for profile driven optimization (PDO).

GPS [95] is a hardware-software solution that maintains duplicate physical replicas of the shared memory pages in the local memory of each GPU, updates these replicas via proactive stores and employs a dynamic unsubscription technique to eliminate the unwanted replicas. While GPS improves interconnect efficiency by eliminating stores to GPUs that do not access the pages, which is similar in spirit to FINEPACK's write coalescing. GPS does nothing to improve the network efficiency for small transfers however and requires invasive changes to the GPU's memory system. In GPS transfers happen at a cacheline granularity and results in unneeded bytes traversing the interconnect if not all bytes within a cacheblock are updated by the application. It also requires programmer effort in some cases to identify the profiling region as the automatic subscription is valid only for iterative applications.

We compared FINEPACK versus GPS for individual applications that had the same reference implementation. We see that for applications where the subscription benefits in GPS are able to offset the low efficiency due to unneeded transfers within a cacheline, GPS performs well. In other workloads where these unneeded transfers result in larger inefficiencies, FINEPACK performs better than GPS, with no need for a new GPU-centric memory subscription mechanism. On average, across our evaluated workloads we find FINEPACK is 10% slower than GPS on a 4-GPU PCIe system, however FINEPACK is a

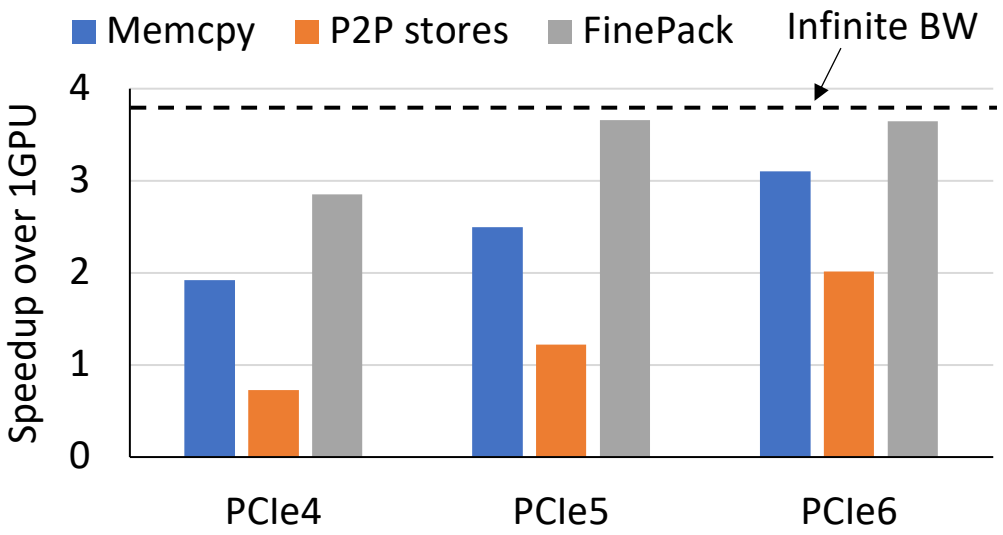


Figure 5.11: Performance sensitivity to interconnect bandwidth

vastly simpler modification to a GPU's architecture and remains fully transparent to the programmer and application without need for special libraries or API usage.

Applicability to other interconnect architectures: Similar to PCIe, other multi-GPU interconnects such NVLink and Infiniband also implement packetized transfers over the interconnect. Though the exact fields in the packet headers for other interconnect architectures are similar but never the same to PCIe, we still expect packets routed between the same source-destination pair to contain configuration fields that can be transferred once, similar to FINEPACK, instead of being transferred for every store, thus conserving interconnect bandwidth in most protocols - albeit at differing rates. We note that the spatial locality fundamentally exploited by FINEPACK to perform address compression is agnostic to the interconnect architecture being used. Hence, though our implementation considered PCIe since it is an open standard, we expect FINEPACK to provide significant performance gains for other interconnect architectures as well. For interconnects such as Infiniband that support larger packet sizes than PCIe, we expect even similar or greater benefit from FINEPACK's ability to aggregate small transfers into efficient bulk transfers.

Alternate FINEPACK implementations: In addition to the proposed design, we per-

formed opportunity studies to evaluate an alternate design where the protocol overheads are minimized by separating out the base address and other common fields into a special *configuration packet*. This packet will define the packet characteristics for the store-packets that follow it until the next configuration packet is sent in a stateful manner. In principle, this is similar to *virtual circuit networks* where the path taken by the first packet is assigned to all packets succeeding it. However, analytical modeling showed that the protocol overhead reduction possible in such a design is not as efficient as the current FINEPACK design. FINEPACK packs multiple stores as sub-transactions within a single PCIe packet, thus requiring one common sequence number, ECRC and LCRC per packet. In contrast, for this competing design, though the common header fields across multiple stores are sent as a single configuration packet, the stores that succeed it must still be independent PCIe packets minimally requiring their own sequence number and CRC fields. This results in an at least additional 10-byte overhead per store when compared to FINEPACK. Thus, for a packet containing 32-64 stores (FINEPACK can typically coalesce 38 stores before emitting a packet), this alternate design is approximately 18% less efficient than FINEPACK. This type of designs will also require significant modifications to the existing PCIe layer because the packet structure for the store-packets will not be compatible with the PCIe packets. We believe FINEPACK is superior to this alternative.

Scaling beyond 4 GPUs: As the number of GPUs in the system increases, the SRAM storage required for FINEPACK's remote write queue will also increase per addressable peer GPU in the system. If the store buffer size becomes a first order design constraint (note that GPU L2 caches are tens of MB in recent GPUs), the size of the per GPU buffer could be reduced to limit the number of entries. The impact of reducing the maximum coalescing size is left for future work however. At larger GPU counts, we expect both the average store size to remain similar to that shown in Fig. 4 and the address locality that is exploited by FINEPACK to remain largely intact because scaling an application across GPU counts does not change the address access patterns within the small time windows that FINEPACK

operates.

5.6 Related work

Small message aggregation: Aggregation of small messages to reduce overheads and improve network performance has been proposed both at software and hardware level. There are a variety of software approaches, ranging from runtimes [160, 91], compilers [5], or application level [23]. The typical approach involves aggregating messages targeting the same end node and/or compressing the final aggregation buffer before it is sent. All these approaches suffer from non negligible software overheads and incur extra transmission latencies. For this reason they typically require additional mitigation mechanisms (such as multi-threading) or limit their applicability to specific domains. A particular successful use case of software based aggregation is the acceleration of collective communication [64].

Network level approaches [61, 26], instead are designed as part of the routers and aggregate messages as they travel inside the network. They are transparent to the user and the applications and provide a net benefit because the hardware can inexpensively monitor the effectiveness of the aggregation and use it as needed.

In this paper we also propose a hardware based approach which is transparent to the application, but also to the network itself as it logically sits at the interface between the GPU and the network ingress/egress ports. Our approach is based on the aggregation memory references following a scheme that is reminiscent of that found inside TLBs [169, 132, 90] as a way improve virtual address space coverage and reduce TLB footprint. While the end goal of TLB range compressors and our approach is different, the core idea of exploiting memory address locality to reduce space is relatively similar.

Multi-GPU performance Prior work [7, 89, 171, 17, 11, 68] has explored different mechanisms both at the hardware and software levels to improve performance in multi-GPU systems. Our work explores optimizing small access efficiency thereby improving strong scaling in multi-GPU systems. Prior work [150, 129, 130] has also proposed techniques

for inter-GPU coherence. FINEPACK avoids an expensive coherence protocol despite maintaining GPU memory model compatibility.

Several works propose multi-GPU memory management solutions. Griffin [11] optimizes page migration to improve multi-GPU performance. CARVE [171] caches remote data in local DRAM to improve locality. PROACT [97] proposes a software framework to aggregate fine-grained transfers and improve interconnect efficiency, however it requires significant effort from the programmer. GPS [95] adopts a publish-subscribe paradigm to improve multi-GPU memory management but requires invasive changes to the memory system. FINEPACK on the other hand, optimizes fine-grained transfers and improves performance while being transparent to the application as well as the GPU memory system.

NUMA memory management: Dashti et al. [32] develop a Linux memory placement algorithm for mitigating address traffic congestion in NUMA systems. Works [173, 1, 75, 128, 38] perform NUMA-aware optimizations. Works also explored hardware-based peer caching [83, 126, 31, 143, 12].

5.7 Conclusion

Performing fine grain writes to other GPU memories is a natural paradigm for programming multi-GPU systems. Unfortunately due to interconnect inefficiency, GPU programmers today are forced to either suffer through poor performance when using P2P stores, or artificially aggregate data updates into program phases that kill communication and computation overlap, but improve interconnect efficiency. FinePack provides the best of both worlds through repacketization of fine-grained transfers to eliminate protocol overhead and aggressive coalescing to reduce redundant data transferred on the interconnect. By leveraging locality inherent in the physical address stream egressing single source to single destination GPUs, FinePack is able to achieve 5× efficiency improvement over peer-to-peer stores that translates into an average speedup of 3.9× over peer-to-peer stores.

CHAPTER VI

Conclusion

Suboptimal management of memory and bandwidth is one of the primary causes of low performance on systems comprising multiple GPUs. Existing multi-GPU systems require hardware and software enhancements to improve interconnect efficiency and achieve good strong scaling performance.

In this work, we proposed PROACT, a hardware and software system that improves multi-GPU performance by overcoming the limitations of existing inter-GPU communication mechanisms. PROACT combines the advantages of peer-to-peer transfers with that of bulk transfers to enable interconnect friendly data transfers while hiding transfer latencies. Demonstrated across three different 4-GPU system architectures, PROACT provides an average speedup of 3× over a single-GPU implementation, capturing 83% of the theoretical limit. We also show how PROACT provides dramatic scalability improvements on next generation systems with large numbers of GPUs, achieving up to 12× speedup over a single-GPU implementation, while capturing 77% of the available opportunity. Scalable multi-GPU programming is no easy task; to maximize programmer productivity, runtime systems like PROACT will be necessary to enable rapid development cycles while leveraging next generation architectural improvements in future GPUs.

Next, we reviewed and evaluated GPS, a set of architectural enhancements 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 all subscribers. This enables the subscribers to read data from local memory at high bandwidth. GPS provides these advantages while retaining compatibility with the conventional GPU programming and memory models. Evaluated on a model of 4 NVIDIA V100 GPUs connected by varying interconnect architectures, GPS offers a speedup of up to 3.8× (2.5× on average) over 1 GPU and performs 3.3× better than the next best available multi-GPU programming paradigm.

Finally, we elaborated our work to improve the interconnect efficiency of small accesses in multi-GPU systems. Performing fine grain writes to other GPU memories is a natural paradigm for programming multi-GPU systems. Unfortunately due to interconnect inefficiency, GPU programmers today are forced to either suffer through poor performance when using P2P stores, or artificially aggregate data updates into program phases that kill communication and computation overlap, but improve interconnect efficiency. FINEPACK provides the best of both worlds through repacketization of fine-grained transfers to eliminate protocol overhead and aggressive coalescing to reduce redundant data transferred on the interconnect. By leveraging locality inherent in the physical address stream egressing single source to single destination GPUs, FINEPACK is able to achieve 5× efficiency improvement over peer-to-peer stores that translates into an average speedup of 3.9× over peer-to-peer stores.

Strong scaling in multi-GPU systems is a challenging task. This dissertation has shown that building blocks such as PROACT, GPS and FINEPACK can successfully enable GPU programmers to perform inter-GPU transfers efficiently with minimal programming overhead and high interconnect efficiency, thereby improving the overall system performance. We hope this dissertation inspires a new body of research on differentiating the multi-GPU programming paradigm from single GPU programming semantics to help extract the best possible performance out of next generation multi-GPU systems.

BIBLIOGRAPHY

BIBLIOGRAPHY

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

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

[3] Jasmin Ajanovic. PCI Express 3.0 Overview. In *Proceedings of Hot Chip: A Symposium on High Performance Chips*, 2009.

[4] Erik Alerstam, William Chun Yip Lo, Tianyi David Han, Jonathan Rose, Stefan Andersson-Engels, and Lothar Lilge. Next-generation Acceleration and Code Optimization for Light Transport in Turbid Media using GPUs. *Biomedical optics express*, 1(2):658–675, 2010.

[5] Michail Alvanos, Montse Farreras, Ettore Tiotto, and Xavier Martorell. Automatic communication coalescing for irregular computations in upc language. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, pages 220–234, 2012.

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

[7] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 320–332. ACM, 2017.

[8] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 416–427. IEEE Computer Society, 2012.

[9] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E Strom, and Daniel C Sturman. An Efficient Multicast Protocol for Content-Based

Publish-Subscribe Systems. In *International Conference on Distributed Computing Systems*, 1999.

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

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

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

[13] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 12. ACM, 2011.

[14] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. 'zeppelin': An soc for multichip architectures. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 40–42. IEEE, 2018.

[15] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri 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*, page 19. ACM, 2015.

[16] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *ACM SIGPLAN Notices*, volume 52, pages 235–248. ACM, 2017.

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

[18] Tarun Beri, Sorav Bansal, and Subodh Kumar. The Unicorn Runtime: Efficient Distributed Shared Memory Programming for Hybrid CPU-GPU Clusters. *IEEE Trans. Parallel Distrib. Syst.*, 28(5):1518–1534, May 2017.

[19] Abhishek Bhattacharjee and Margaret Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 290–301. ACM, 2009.

[20] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics (TOG)*, 22(3):917–924, 2003.

[21] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-accelerated Database Systems: Survey and Open Challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[22] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B Jablin, Nacho Navarro, and Wen-mei W Hwu. Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 3–13. ACM, 2015.

[23] Vito Giovanni Castellana, Maurizio Drocco, John Feo, Jesun Firoz, Thejaka Kanewala, Andrew Lumsdaine, Joseph Manzano, Andrés Marquez, Marco Minutoli, Joshua Suetterlein, et al. A parallel graph environment for real-world data analytics workflows. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1313–1318. IEEE, 2019.

[24] John Cavazos and Michael FP O'Boyle. Automatic Tuning of Inlining Heuristics. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 14–14. IEEE, 2005.

[25] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 13. ACM, 2011.

[26] Dong Chen, Noel Eisley, Philip Heidelberger, Sameer Kumar, Amith Mamidala, Fabrizio Petrini, Robert Senger, Yutaka Sugawara, Robert Walkup, Burkhard Steinmacher-Burow, et al. Looking under the hood of the ibm blue gene/q network. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[27] Jianmin Chen, Zhuo Huang, Feiqi Su, Jih-Kwon Peir, Jeff Ho, and Lu Peng. Weak Execution Ordering-exploiting Iterative Methods on Many-core GPUs. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 154–163. IEEE, 2010.

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

[29] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[30] Andrew Corrigan, Fernando F Camelli, Rainald Löhner, and John Wallin. Running Unstructured Grid-based CFD Solvers on Modern Graphics Hardware. *International Journal for Numerical Methods in Fluids*, 66(2):221–229, 2011.

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

[32] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA systems. *ACM SIGPLAN Notices*, 48(4), 2013.

[33] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[34] Andrew Davidson and John Owens. Toward Techniques for Auto-tuning GPU Algorithms. In *International Workshop on Applied Parallel Computing*, pages 110–119. Springer, 2010.

[35] Timothy A Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[36] Glenn Dearth and Vyas Venkatraman. Inside DGX-2, http://on-demand.gputechconf.com/gtc/2018/presentation/s8688-extending-the-connectivity-and-reach-of-the-gpu.pdf.

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

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

[39] Anders Eklund, Mats Andersson, and Hans Knutsson. fMRI Analysis on the GPU—Possibilities and Challenges. In *Computer methods and programs in biomedicine*, volume 105, pages 145–161. Elsevier, 2012.

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

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

[42] Benedict R Gaster. HSA Memory Model. In *Hot Chips Symposium*, 2013.

[43] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven HM Buijssen, Matthias Grajewski, and Stefan Turek. Exploring Weak Scalability for FEM Calculations on a GPU-enhanced Cluster. *Parallel Computing*, 33(10):685–699, 2007.

[44] Dominik Goddeke, Hilmar Wobker, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, and Stefan Turek. Co-processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FEASTGPU. *International Journal of Computational Science and Engineering*, 4(4):254–269, 2009.

[45] Brice Goglin. Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX. In *2008 IEEE International Conference on Cluster Computing*, pages 223–231. IEEE, 2008.

[46] Ping Guo and Liqiang Wang. Auto-tuning CUDA Parameters for Sparse Matrix-vector Multiplication on GPUs. In *Computational and Information Sciences (ICCIS), 2010 International Conference on*, pages 1154–1157. IEEE, 2010.

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

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

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

[50] Mark D Hill, James R Larus, Steven K Reinhardt, and David A Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 11(4):300–318, 1993.

[51] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

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

[53] Haibo Hu, Dik Lun Lee, and Victor Lee. Distance Indexing on Road Networks. In *Proceedings of the 32nd international conference on Very large data bases*, pages 894–905. VLDB Endowment, 2006.

[54] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[55] Bruce M Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2(1):5–32, 1970.

[56] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. Automatic CPU-GPU Communication Management and Optimization. In *ACM SIGPLAN Notices*, volume 46, pages 142–151. ACM, 2011.

[57] B. Jang, D. Kaeli, S. Do, and H. Pien. Multi GPU implementation of iterative tomographic reconstruction algorithms. In *Proc. IEEE Intl. Symp. Biomed. Imag.*, pages 185–8, 2009.

[58] Sylvain Jeaugey. NCCL 2.0. *GTC*, 2017.

[59] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[60] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, David E Shaw, John Kim, and William J Dally. A Detailed and Flexible Cycle-Accurate Network-On-Chip Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[61] Nan Jiang, Larry Dennison, and William J Dally. Network endpoint congestion control for fine-grained communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[62] Stephen Jones. Introduction to Dynamic Parallelism. In *GPU Technology Conference Presentation S*, volume 338, page 2012, 2012.

[63] Stephen Jones. Introduction to Dynamic Parallelism. In *GPU Technology Conference Presentation S*, volume 338, page 2012, 2012.

[64] Laxmikant V Kalé, Sameer Kumar, and Krishnan Varadarajan. A framework for collective personalized communication. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003.

[65] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An Auto-tuning Framework for Parallel Multicore Stencil Computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[66] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[67] D Kim, D Pal, J-B Thibault, and Jeffrey A Fessler. Improved ordered subsets algorithm for 3D X-ray CT image reconstruction. *Proc. 2nd Intl. Mtg. on image formation in X-ray CT*, pages 378–81, 2012.

[68] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1357–1370, 2020.

[69] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2016.

[70] Jens Krüger and Rüdiger Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In *ACM SIGGRAPH 2005 Courses*, page 234. ACM, 2005.

[71] Scott Robert Ladd. Acovea: Analysis of Compiler Options Via Evolutionary Algorithm. *Describing the EvolutionaryAlgorithm", http://stderr. org/doc/acovea/html/acoveaga. html*, 2007.

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

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

[74] Shin-Ying Lee and Carole-Jean Wu. CAWS: Criticality-aware Warp Scheduling for GPGPU workloads. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 175–186. ACM, 2014.

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

[76] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[77] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *International Symposium on Workload Characterization (IISWC)*, 2018. `https://github.com/uuudown/Tartan`, last accessed on 08/17/2020.

[78] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. SMAT: an Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication. In *ACM SIGPLAN Notices*, volume 48, pages 117–126. ACM, 2013.

[79] Yinan Li, Jack Dongarra, and Stanimire Tomov. A Note on Auto-tuning GEMM for GPUs. In *International Conference on Computational Science*, pages 884–892. Springer, 2009.

[80] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008.

[81] Rui Liu, Lin Fu, Bruno De Man, and Hengyong Yu. GPU-based Branchless Distance-Driven Projection and Backprojection. *IEEE Transactions on Computational Imaging*, 2017.

[82] Yong Long, Jeffrey A Fessler, and James M Balter. 3D Forward and Back-projection for X-ray CT using Separable Footprints. *IEEE transactions on medical imaging*, 29(11):1839–1850, 2010.

[83] Daniel Lustig and Margaret Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 354–365. IEEE, 2013.

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

[85] Henry Massalin. Superoptimizer: A Look at the Smallest Program. In *ACM SIGPLAN Notices*, volume 22, pages 122–126. IEEE Computer Society Press, 1987.

[86] M. McGaffin and J. A. Fessler. Alternating dual updates algorithm for X-ray CT reconstruction on the GPU. *IEEE Trans. Computational Imaging*, 1(3):186–99, September 2015.

[87] Madison G McGaffin and Jeffrey A Fessler. Alternating Dual Updates Algorithm for X-ray CT Reconstruction on the GPU. *IEEE transactions on computational imaging*, 1(3):186–199, 2015.

[88] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. Beyond the Socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–135, 2017.

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

[90] Sparsh Mittal. A survey of techniques for architecting tlbs. *Concurrency and computation: practice and experience*, 29(10):e4061, 2017.

[91] Alessandro Morari, Oreste Villa, Antonino Tumeo, Daniel Chavarria Miranda, and Mateo Valero Cortés. Gmt: Enabling easy development and efficient execution of irregular applications on commodity clusters. In *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC13: Denver, CO, 2013: Technical program posters*, pages 1–2. Association for Computing Machinery (ACM), 2013.

[92] J Eliot B Moss, Paul E Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla E Brodley, and David Scheeff. Learning to Schedule Straight-line Code. In *Advances in Neural Information Processing Systems*, pages 929–935, 1998.

[93] K. Mueller, F. Xu, and N. Neophytou. Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography? In *Proc. SPIE 6498 Comp. Imag.*, page 64980N, 2007.

[94] Gero Mühl. *Large-Scale Content-Based Publish-Subscribe Systems*. PhD thesis, Technische Universität Darmstadt, 2002.

[95] Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. Gps: A global publish-subscribe model for multi-gpu memory management. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–58, 2021.

[96] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, David Nellans, and Thomas Wenisch. Finepack: Improving the efficiency of fine-grainedtransfers in multi-gpu systems. In *Under submission International Symposium on Computer Architecture (ISCA)*, 2021.

[97] Harini Muthukrishnan, David Nellans, Daniel Lustig, Jeffrey Fessler, and Thomas Wenisch. Efficient multi-gpu shared memory via automatic optimization of fine-grained transfers. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2021.

[98] Harini Muthukrishnan, Thomas F Wenisch, and Jeffrey A Fessler. Improving gpu scaling for x-ray ct. In *Proc. of 5th international conference on image formation in X-ray computed tomography*, 2018.

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

[100] J. Ni, X. Li, T. He, and G. Wang. Review of parallel computing techniques for computed tomography image reconstruction. *Current Medical Imaging Reviews*, 2(4):405–14, November 2006.

[101] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[102] Kyle E Niemeyer and Chih-Jen Sung. Recent Progress and Challenges in Exploiting Graphics Processors in Computational Fluid Dynamics. In *The Journal of Supercomputing*, volume 67, pages 528–564. Springer, 2014.

[103] Cedric Nugteren and Valeriu Codreanu. CLTune: A Generic Auto-tuner for OpenCL Kernels. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 195–202. IEEE, 2015.

[104] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT Library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 30. ACM, 2009.

[105] Nvidia. NVIDIA DGX-1 Essential Instrument of AI Research, https://www.nvidia.com/en-us/data-center/dgx-1/.

[106] Nvidia. NVIDIA Tesla GPU Accelerators, https://www.nvidia.com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf.

[107] Nvidia. NVIDIA Tesla P100 Whitepaper, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[108] Nvidia. NVIDIA Tesla V100 Datasheet, http://www.nvidia.com/content/pdf/volta-datasheet.pdf.

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

[110] Nvidia. Unified Memory in CUDA 6, https://devblogs.nvidia.com/unified-memory-in-cuda-6/, 2013.

[111] NVIDIA. NVIDIA CUDA Compiler Driver NVCC, http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html, 2018.

[112] Nvidia. NVIDIA DGX-2, Datasheet, http://images.nvidia.com/content/pdf/dgx2-print-datasheet-a4-nv-620850-r9-web.pdf, 2018.

[113] Nvidia. NVIDIA TESLA V100 GPU Accelerator, https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf, 2018.

[114] Nvidia. NVLINK Fabric for Advanced Multi-GPU Processing, https://www.nvidia.com/en-us/data-center/nvlink/, 2018.

[115] Nvidia. NVSwitch: Leveraging NVLink to Maximum Effect, https://news.developer.nvidia.com/nvswitch-leveraging-nvlink-to-maximum-effect/, 2018.

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

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

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

[119] Marc S Orr, Shuai Che, Ayse Yilmazer, Bradford M Beckmann, Mark D Hill, and David A Wood. Synchronization Using Remote-scope Promotion. *Computer Architecture News*, 43(1), 2015.

[120] Gregory F Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[121] Usman Pirzada. Intel hints towards an xe "coherent multi-gpu" future with cxl interconnect, 2019. https://wccftech.com/intel-xe-coherent-multi-gpu-cxl/, last accessed on 10/28/2021.

[122] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. Efficient Inter-node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 80–89. IEEE, 2013.

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

[124] Sreeram Potluri, Nathan Luehr, and Nikolay Sakharnykh. Simplifying Multi-GPU Communication with NVSHMEM. In *GPU Technology Conference*, 2016.

[125] Sreeram Potluri, Hao Wang, Devendar Bureddy, Ashish Kumar Singh, Carlos Rosales, and Dhabaleswar K Panda. Optimizing MPI Communication on Multi-GPU Systems using CUDA Inter-process Communication. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1848–1857. IEEE, 2012.

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

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

[128] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *The Proceedings of the VLDB Endowment (PVLDB)*, 8, 2015.

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

[130] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 582–595. IEEE, 2020.

[131] Steve Rennich. Cuda C/C++ Streams and Concurrency. In *GPU Technology Conference*, 2011.

[132] Theodore H Romer, Wayne H Ohlrich, Anna R Karlin, and Brian N Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 176–187, 1995.

[133] Jeffrey M Rosen, Junjie Wu, TF Wenisch, and JA Fessler. Iterative Helical CT Reconstruction in the Cloud for Ten Dollars in Five Minutes. In *Proc. Intl. Mtg. on Fully 3D Image Recon. in Rad. and Nuc. Med*, pages 241–4, 2013.

[134] Amit Sabne, Xiao Wang, Sherman J Kisner, Charles A Bouman, Anand Raghunathan, and Samuel P Midkiff. Model-based Iterative CT Image Reconstruction on GPUs. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–220. ACM, 2017.

[135] Romelia Salomon-Ferrer, Andreas W Gotz, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. *Journal of chemical theory and computation*, 9(9):3878–3888, 2013.

[136] R. Sampson, M. G. McGaffin, T. F. Wenisch, and J. A. Fessler. Investigating multithreaded SIMD for helical CT reconstruction on a CPU. In *Proc. 4th Intl. Mtg. on image formation in X-ray CT*, pages 275–8, 2016.

[137] Dana Schaa and David Kaeli. Exploring the Multiple-GPU Design Space. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[138] Tim C Schroeder. Peer-to-peer and Unified Virtual Addressing. In *GPU Technology Conference, NVIDIA*, 2011.

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

[140] PCI SIG. PCI Express Base Specification Revision 3.1a Specifications, 2010.

[141] PCI SIG. PCI Express Base Specification Revision 3.1a Specifications, 2010.

[142] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. Efficient GPU synchronization without scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 647–659, 2015.

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

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

[145] Joe Stam. Maximizing GPU Efficiency in Extreme Throughput Applications. In *Proceedings of the GPU Technology Conference 2009*, 2009.

[146] Mark William Stephenson. *Automating the Construction of a Complier Heuristics using Machine Learning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[147] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Matrix Multiplication Beyond Auto-tuning: Rewrite-based GPU Code Generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 15:1–15:10, New York, NY, USA, 2016. ACM.

[148] Gregory M Striemer and Ali Akoglu. Sequence Alignment with GPU: Performance and Design Challenges. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[149] Mark Sutherland, Joshua San Miguel, and Natalie Enright Jerger. Texture Cache Approximation on GPUs. In *Workshop on Approximate Computing Across the Stack*, 2015.

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

[151] Gábor Takács and Domonkos Tikk. Alternating least squares for personalized ranking. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 83–90, 2012.

[152] J-B. Thibault, K. Sauer, C. Bouman, and J. Hsieh. A three-dimensional statistical approach to improved image quality for multi-slice helical CT. *Med. Phys.*, 34(11):4526–44, November 2007.

[153] Ivan S Ufimtsev and Todd J Martinez. Strategies for Two-electron Integral Evaluation. *Journal of Chemical Theory and Computation*, 4(2):222–231, 2008.

[154] Stephen Van Doren. Hoti 2019: Compute express link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18. IEEE, 2019.

[155] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

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

[157] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.

[158] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *Computer Science-Research and Development*, 26(3-4):257, 2011.

[159] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A High-performance Graph Processing Library on the GPU. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016.

[160] Lukasz Wesolowski, Ramprasad Venkataraman, Abhishek Gupta, Jae-Seung Yeom, Keith Bisset, Yanhua Sun, Pritish Jetley, Thomas R Quinn, and Laxmikant V Kale. Tram: Optimizing fine-grained communication with topological routing and aggregation of messages. In *2014 43rd International Conference on Parallel Processing*, pages 211–220. IEEE, 2014.

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

[162] WikiChip. Infinity Fabric (IF) - AMD , 2019. `https://en.wikichip.org/wiki/amd/infinity_fabric`, last accessed on 11/23/2020.

[163] Wikipedia. PCI Express, 2005. `https://en.wikipedia.org/wiki/PCI_Express`, last accessed on 10/28/2021.

[164] Wikipedia. Band Matrix, https://en.wikipedia.org/wiki/band_matrix, 2018.

[165] Wikipedia. Jacobi Method, https://en.wikipedia.org/wiki/jacobi_method, 2018.

[166] Wikipedia. PageRank, https://en.wikipedia.org/wiki/pagerank, 2018.

[167] Shucai Xiao and Wu-chun Feng. Inter-block GPU Communication via Fast Barrier Synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[168] Qingang Xiong, Bo Li, Feiguo Chen, Jingsen Ma, Wei Ge, and Jinghai Li. Direct Numerical Simulation of Sub-grid Structures in Gas–solid Flow—GPU Implementation of Macro-scale Pseudo-particle Modeling. *Chemical Engineering Science*, 65(19):5356–5365, 2010.

[169] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 698–710, 2019.

[170] Erlin Yao, Yungang Bao, Guangming Tan, and Mingyu Chen. Extending Amdahl's Law in the Multicore Era. *ACM SIGMETRICS Performance Evaluation Review*, 37(2):24–26, 2009.

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

[172] K. Zeng, E. Bai, and G. Wang. A fast CT reconstruction scheme for a general multi-core PC. *Intl. J. Biomedical Im.*, 2007:29160, 2007.

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

[174] Yongpeng Zhang and Frank Mueller. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 155–164. ACM, 2012.

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

[176] T. Zinßer and B. Keck. Systematic performance optimization of cone-beam backprojection on the Kepler architecture. In *Proc. Intl. Mtg. on Fully 3D Image Recon. in Rad. and Nuc. Med*, pages 225–8, 2013.