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

Harini Muthukrishnan[1][2], Daniel Lustig[1], Oreste Villa[1], Thomas Wenisch[2], and David Nellans[1]

[1]NVIDIA
[2]University of Michigan

***Abstract*—Recent studies have shown that using fine-grained peer-to-peer (P2P) stores to communicate among devices in multi-GPU systems is a promising path to achieve strong performance scaling. In many irregular applications, such as graph algorithms and sparse linear algebra, small sub-cache line (4-32B) stores arise naturally when using the P2P paradigm. This is particularly problematic in multi-GPU systems because inter-GPU interconnects are optimized for bulk transfers rather than small operations. As a consequence, application developers either resort to complex programming techniques to work around this small transfer inefficiency or fall back to bulk inter-GPU DMA transfers that have limited performance scalability. We propose FinePack, a set of limited I/O interconnect and GPU hardware enhancements that enable small peer-to-peer stores to achieve interconnect efficiency that rivals bulk transfers while maintaining the simplicity of a peer-to-peer memory access programming model. Exploiting the GPU's weak memory model, FinePack dynamically coalesces and compresses small writes into a larger I/O message that reduces link-level protocol overhead. FinePack is fully transparent to software and requires no changes to the GPU's virtual memory system. We evaluate FinePack on a system comprising 4 Volta GPUs on a PCIe 4.0 interconnect to show FinePack improves interconnect efficiency for small peer-to-peer stores by 3×. This results in 4-GPU strong scaling performance 1.4× better than traditional DMA based multi-GPU programming and comes within 71% of the maximum achievable strong scaling performance.**

I. Introduction

As demand for compute throughput skyrockets, GPU vendors continue to deliver larger systems comprising more GPUs, additional 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 computing systems grow. Recently, tasks such DL training with billions of parameters have been scaled to systems with thousands of GPUs thanks to continuous development effort among the GPU computing industry [16], [17].

However, *strong scaling* problems across increasing numbers of GPUs is 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 today's best GPU-to-GPU interconnects deliver bandwidth that is an order of magnitude smaller than the GPU's locally attached memory bandwidth [22], [33]. This inter-GPU bottleneck is so severe that many multi-GPU programs exhibit slowdowns when compared to their single GPU counterparts, even after tedious manual optimization [22], [31], [32].

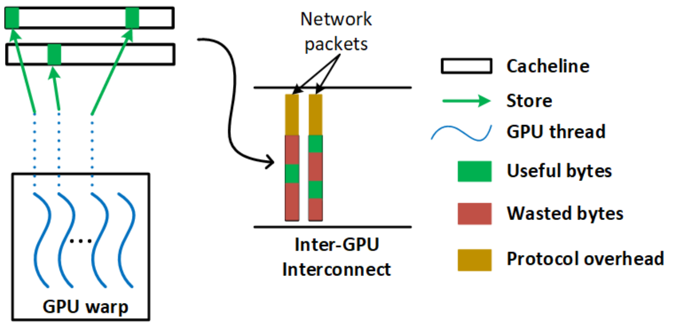


Fig. 1: A GPU's SM issues writes at sub-cacheline granularity for many irregular applications. No coalescing beyond the L1 cache results in transfer inefficiencies, due to protocol overhead and unread bytes at the receiver.

To improve multi-GPU strong scaling performance, multi-GPU programming models and architectural enhancements have been an area of active research and development over the past few years [4], [31], [32], [55]. Recent work has shown the benefits of programming paradigms built around replicating data structures across GPUs and performing updates via proactive peer-to-peer (P2P) stores [31], [32]. Such paradigms enable reads to be performed locally during computation, eliminating inter-GPU transfers from the critical path. Proactive peer-to-peer stores are efficient, as they do not stall the issuing GPU core. They provide a natural overlap of program compute and communication, which is necessary to keep the GPU compute cores fully occupied. P2P stores also maintain a familiar shared-memory programming model and fall within the standard GPU memory consistency model, thus maintaining easy programmability when porting applications from one to many GPUs.

However, as shown in Figure 1, the primary drawback to programming with peer-to-peer stores is that it results in transfers at sub-cacheline granularity for irregular applications. We profiled multiple applications (described in Section V) and found that on average over 63% of inter-GPU transfers initiated by P2P stores carry a payload smaller than 32B; drastically smaller than the typical granularity of DMA op-

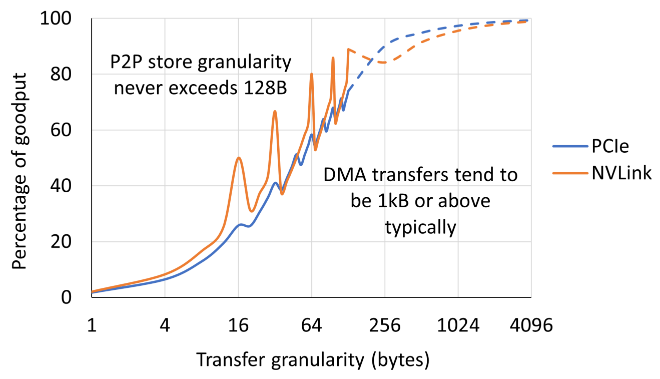


Fig. 2: Percentage of useful bytes transferred vs. maximum theoretical throughput, when varying the transfer size of peer-to-peer stores. Measured in real systems up to 128B and projected beyond because peer-to-peer stores never exceed 128B.

erations in GPU programs. This is mainly because stores in such applications naturally scatter across different cache lines, and remote stores do not undergo coalescing beyond L1, as explained later in Section III.

Figure 2 shows the P2P store goodput (number of data payload bytes divided by total bytes sent) of two common GPU interconnect protocols, measured in real systems for transfers up to 128B and then projected beyond[1]. Small P2P stores that traverse the inter-GPU interconnect achieve poor interconnect efficiency compared to full cache line (128B) transfers and even more with respect 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 gain can be more than offset by the decrease in interconnect goodput, resulting in a net performance loss. Thus, an ideal scalable multi-GPU system should support both a programming model that enables overlap of compute and communication while also achieving high interconnect efficiency for both small and large transfers alike.

We propose FinePack, a set of GPU architectural enhancements that with minimal embedded interconnect protocol support can improve the interconnect efficiency of fine-grained peer-to-peer store operations for GPUs. By exploiting the GPU's weak memory model and the application's spatio-temporal locality at larger memory ranges (MBs-GBs), we show that it is possible to dramatically improve small transfer efficiency without requiring any changes to the programmer's software interface for peer-to-peer stores. 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 base+offset compression to minimize framing and link-level protocol overhead. On the receiving side, 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 inter-GPU interconnect is changed. All changes remain fully transparent to the programmer. Through coalescing and compression, FinePack drastically decreases both the protocol overheads of small writes and wasted bandwidth arising from repeated issue of remote stores to the same address.

Although small message aggregation has been previously proposed in the context of HPC network optimization [37], this work advances state of the art in multi-GPU systems and interconnects with the following contributions:

- We identify the need for improving the efficiency of small granularity stores with hardware support and show that existing and emerging interconnects do not handle them appropriately.
- We propose minor modifications to the existing link-level PCIe interconnect protocol to significantly improve small store efficiency. Our approach can also be applied to other interconnect protocols.
- We design GPU architectural enhancements to deploy FinePack while remaining fully transparent to the existing GPU software interface and hardware memory system.
- Through simulation we show that, on a switched PCIe 4.0 interconnect with 4 GPUs, FinePack improves interconnect efficiency by 3×, while providing an average strong scaling performance of 2.4× over a single GPU, capturing 71% of the available opportunity.

II. Background

A. Multi-GPU Memory Management

Page Management: Single-node (i.e., single OS) multi-GPU systems map the memory of all GPUs to a single shared virtual address space. Programmers can perform page management either automatically through APIs such as NVIDIA's Unified Memory (UM) or manually via explicit page allocation, placement, and pinning. While UM offers a convenient way to perform page placement, prior work has shown that despite continued optimization, the cost of migrating pages among GPUs (to optimize locality) is too inefficient to be deployed in multi-GPU systems [1], [31]. Therefore, today most high performance GPU program data structures are allocated on a per-GPU basis and managed explicitly by the programmer, without using UM-based page management. Under this model, page migrations are absent unless explicitly triggered by the programmer.

Data Replication for Locality Management: In multi-GPU systems because a GPU's local memory bandwidth is orders of magnitude higher than to a remote GPU's memory, performing remote reads during computation can stall the compute pipeline and degrade performance. Therefore, particularly for iterative applications, it is common practice

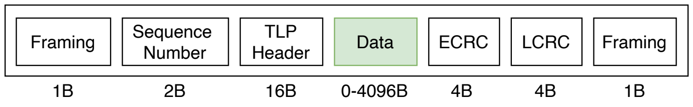
[1]Note that NVLink may or may not send a byte enable flit based on data size and alignment resulting in spikes in its measured goodput [10].

for developers to replicate read-write data structures across multiple GPUs' physical memory and transfer data within the program execution to each GPU, thereby optimizing for memory locality when later reads of this data can occur locally. This way, compute kernels can perform loads from the replica in high bandwidth local memory instead of waiting on slower remote loads, significantly improving multi-GPU performance in most cases.

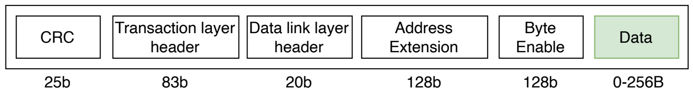
Communication via Proactive Peer-to-Peer Stores: Using replicated data structures requires a logical update protocol implemented by the programmer to synchronize changes among GPUs during each phase of computation. The traditional way to perform such updates is via bulk-synchronous memcpy operations at kernel boundaries; copying whole updated data structures from each producer to the other replicas. However these copies do not often overlap with computation, thus hurting performance. When memory access patterns permit, expert programmers can subdivide kernels and perform memcpy operations in smaller pieces that can be overlapped and for higher efficiency. This approach is challenging even for expert programmers, because it typically requires strict partitioning of the data structures into coarse grained independent pieces.

An alternative data movement approach that is gaining attention involves the GPUs pushing data proactively to all replicas using peer-to-peer stores as soon as the data is generated. This can be done by having the program explicitly not just update its own local copy of the data structure, but also perform duplicate stores directly to the remote replicas. When implemented correctly, GPUs perform loads only to their local copy of each data structure, ensuring good memory bandwidth utilization, while stores are propagated to replicas concurrently with local computation, yet off the critical path of the workload. 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. Though prior work [31], [32] has explored the benefits of fine-grained peer-to-peer stores, they are not used widely due to their performance limitations as described below.

GPU Memory Model and Coherence: The NVIDIA GPU memory consistency model [35] prescribes rules regarding the apparent ordering of GPU memory operations and the values that may be returned by a read operation. GPU memory accesses are weak accesses by default, unless they are atomic operations, fence operations, or they possess explicit synchronization qualifiers. Also, the GPU memory model only requires weak stores to be visible upon performing synchronization or reaching the end of the kernel. Though prior work [41] has proposed hardware coherence protocols for multi-GPU systems, pages owned by a remote GPU are not cached in a GPU's local L2 cache today, so there is no need for inter-GPU coherence traffic. When combined with the visibility requirements for weak stores, this provides FinePack the opportunity to coalesce, aggregate and reorder stores before they reach the interconnect, while maintaining full compatibility with the GPU memory model.



(a) PCIe 4.0



(b) NVLink 1.0

Fig. 3: The packet structure of two common GPU interconnects.

B. Understanding GPU Interconnect Inefficiencies

If peer-to-peer stores are an ideal programming model for multi-GPUs, but have not yet been adopted, we must understand the factors limiting their performance within multi-GPU systems. Because of their early presence in the marketplace, PCIe [50] and NVLink [34] are the predominant interconnects used for multi-GPU systems within a single node today

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

- **Protocol overheads:** Interconnects form network packets by combining a 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 3, 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) will be, as shown in Figure 2.
- **Over-transfer of data:** In situations where the programmer chooses to employ bulk DMA-based transfers at the end of compute kernels and the kernels have performed sparse updates to program data structures, it is difficult for the developer to identify the precise subset of memory locations that were updated. Hence, the programmer ends up transferring unwanted data, i.e., data that was not updated in the compute kernel and/or not consumed by the target GPU. Moreover, DMA-based transfers are typically initiated using software APIs which have to go through many software layers (runtime, driver, etc.) resulting in additional software overheads that become cost prohibitive if the granularity of the data transfer is small.
- **Redundant transfer of data:** When a programmer performs peer-to-peer stores and these stores exhibit temporal locality, many writes to a piece of data may occur before the final value is produced. The value of a given memory location is only known to be final when the program reaches a synchronization barrier under today's weak GPU memory models. Until this barrier

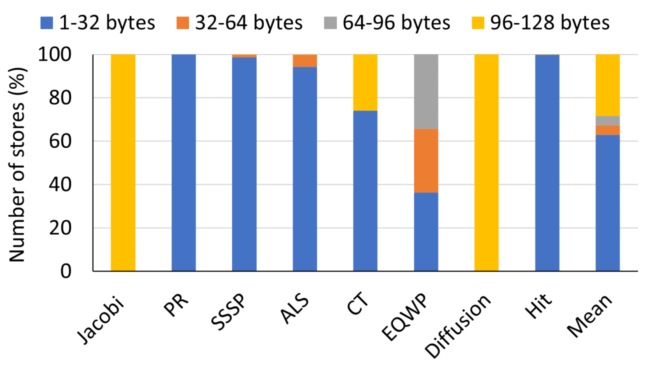


Fig. 4: Average size of remote stores exiting GPU's L1 cache.

occurs, sending multiple P2P stores to the same address is redundant and results in wasted interconnect bandwidth.

- **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.

III. FinePack Motivation

This work makes the important observation that by examining the stream of addresses in packets egressing one GPU on the way to remote GPUs, there is good opportunity for both compressing the packet headers and addresses and coalescing redundant writes before these packets are sent out over the interconnect. In this section, we explain how and why this opportunity arises and how FinePack introduces optimizations that take advantage of it to solve three of the four sources of small packet inefficiency in multi-GPU systems today.

Inter-GPU network packets are formed from stores that originate at the GPU's SM (streaming multiprocessor) core and are destined for memory physically located on another GPU. Each store from an individual GPU thread will range from 1–8B. If there is good spatial *and* temporal locality, the GPU's L1 cache will naturally coalesce memory accesses across a *warp* or *wavefront* of 32 threads into a single memory access of up to 128B[2].

However, if no locality arises due to insufficient optimization on the part of the programmer or the inherent nature of the problem itself (e.g., in graph or sparse matrix algorithms), no coalescing can be performed and accesses to a remote GPU are sent over the interconnect with small payloads of 32B or less. Testing shows that on all recent NVIDIA GPUs, writes to peer GPU memory also do not get cached within the GPU's

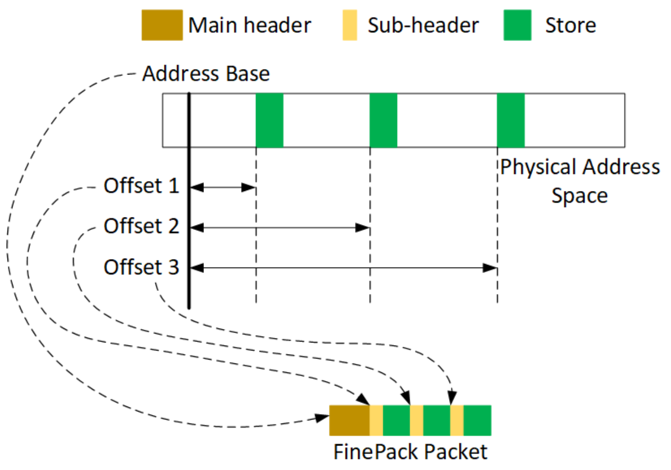


Fig. 5: Spatial locality enables address compression by partitioning the address into a base and offset format.

L2 before egress. This is because the GPU's L2 today is a memory-side cache which is the point of coherence for its locally attached memory. NVIDIA GPUs also do not L2-cache data read remotely from other GPUs because no multi-GPU HW coherence is supported, though this is a subject of some recent academic work [41].

Figure 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 often emit sub-128B communication. Due to the lack of caching or coalescing beyond the L1 cache, these small writes are passed directly to the network. Making things worse is the fact that because physical address spaces today are commonly as large as 48–64b, there is 6–8B of address information required to transfer a full address along with each packet. All of these issues lead to peer-to-peer stores being inefficient on today's GPUs.

Figure 5 illustrates the basic premise on which FinePack is based. When examining a stream of writes egressing a single GPU and destined for a single remote GPU, we observe that these writes exhibit spatial locality when considering limited address ranges (tens of MB to GB). This allows FinePack to combine multiple small writes (to different addresses) that have redundant common address bits into a base plus offset addressing scheme to compress out redundant bits. Furthermore, should any of these writes be destined for the same address, the GPU's weak memory model allows even more efficient compression/coalescing by overwriting the data locally before sending the packet out on the interconnect. Exploiting these two properties via modest architectural modifications allows fine grain P2P stores to achieve an interconnect efficiency that rivals or exceeds that of bulk DMA memory copies and overcomes the primary limitations for adopting P2P stores as the preferred method of achieving strong multi-GPU scaling.

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

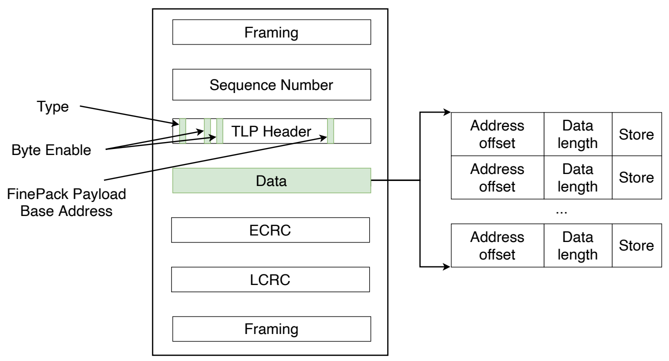


Fig. 6: FinePack packet structure embedded within PCIe. Packet fields modified by FinePack are shown in green.

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 I: PCIe transaction layer protocol (TLP) header fields, as interpreted for FinePack packets. Most fields retain their standard meaning. FinePack-modified fields are shown in bold

	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 II: 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.

IV. FinePack Design

The goal of FinePack is to transfer fine-grained stores across the interconnect more efficiently. It does this by compressing multiple individual transfers into one larger transaction. The FinePack design consists of two primary components: logical extensions to the interconnect transaction layer to define a compressed store operation and the GPU microarchitecture that coalesces transactions in a network-friendly manner. We describe these two components in turn below.

A. FinePack Packet Structure

Figure 6 shows the logical 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 interconnect bandwidth. In effect, FinePack allows stores to share framing and transaction layer header fields across otherwise independent stores.

In FinePack for PCIe, the existing transaction layer protocol (TLP) header fields retain their original meanings and are largely unchanged. However, the payload of our new transaction type now comprises multiple sub-packets concatenated within a single transaction. Each sub-transaction contains a new header that encodes specifics about the sub-packets that differ from the outer transaction, specifically, a compressed address and the size of the sub-packets payload for each individually packed store.

Table I enumerates the fields in the outer PCIe TLP packet[3]. Among the required fields of the PCIe TLP, all fields except the address, data length, and byte enables can be made common for stores having 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-packet and can be used for buffer management by the reciever. The byte enable fields are unused for the FinePack transaction type because the length of each individual write within the FinePack packet has its width specified within the sub-header.

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

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 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 number of bits reserved for addresses and data 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 have swept the header size across different byte counts as shown in Table II. In all cases, ten bits are reserved for the length field (similar to the PCIe protocol) 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 while still allowing a significant addressable range within the boundaries of the outer transaction. A FinePack augmented PCIe implementation consumes buffers and credits the same way a variable length memory write transaction is currently specified on PCIe without change. Also, FinePack optimizes only the

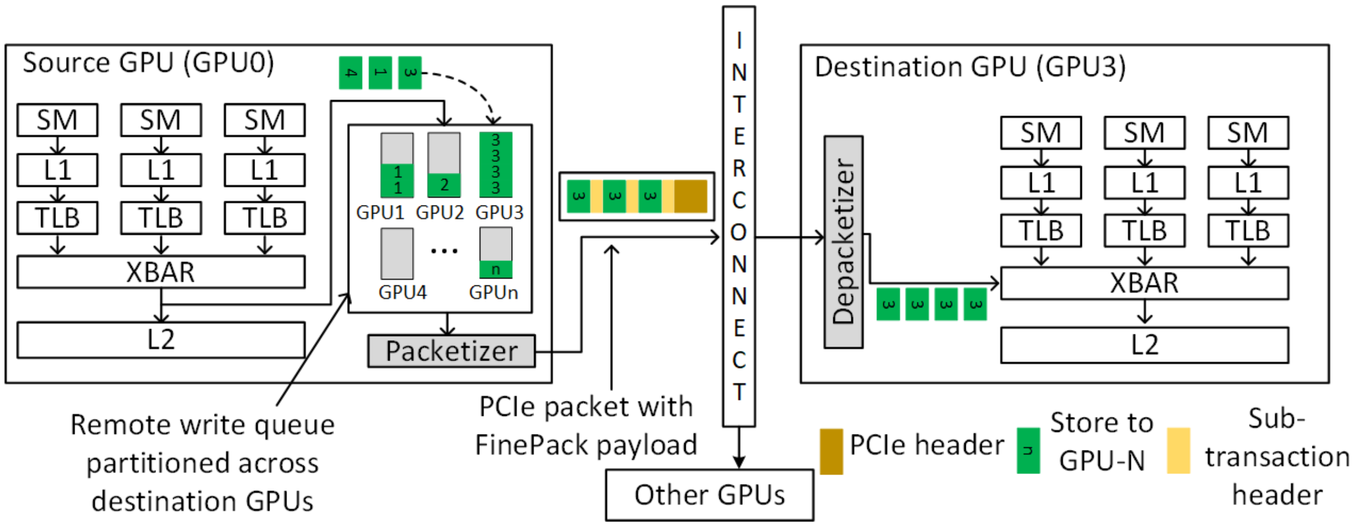


Fig. 7: Overview of the FinePack architecture.

transactions sent by FinePack-aware endpoints in the network and does not impact other PCIe devices connected to switches, or the switches themselves. In FinePack-enabled systems, the CPU's root complex and switches can still send packets over the interconnect unmodified.

B. FinePack Architectural Components

The overall architecture is shown in Figure 7. Our hardware proposal comprises three major components. First is a *remote write queue* that collects outgoing GPU stores at the GPU's network egress port and buffers them in an attempt to combine them into larger transactions. Second, is a *packetizer* to re-packetize the combined stores before sending them to the network using the extended TLP packet format described in Section IV-A. Third, 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 local memory system. We describe each FinePack component in turn below.

Remote Write Queue: The remote write queue is a new dedicated SRAM structure placed between the intra-GPU crossbar and the GPU's network egress port. It buffers outbound remote store transactions such that (1) multiple stores to the same address are overwritten in the queue and only the most recent store to a given address at the time of flushing is transferred to the peer (2) As long as the maximum remote queue size and the maximum PCIe payload size are not exceeded, stores within the address window (determined by the address base and the range of address offset) are accumulated so that the packetizer module can repacketize the stores into the FinePack packet format.

The remote write queue is partitioned across the total number of destination GPUs in the system so that stores targeting each GPU can be coalesced independently. The logical operation of each partition is shown in Figure 8. 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. Separate registers store FinePack configurations, namely, the number of base address bits, number of address offset bits, FinePack sub-header size and the maximum permissible payload size. The sub-header size depends on the number of address offset bits and the maximum payload size, as shown in Table II. An available payload length register is present per partition to keep track of the payload length of the packet to ensure it doesn't exceed the maximum length supported by the interconnect. Thus if the available payload length register is equal to the maximum packet length supported by the interconnect, then the queue partition is empty and if it is zero, then the queue partition is full. The remote write queue also contains counters to keep track of the number of stores present in each partition at a given instant. A separate set of registers also hold the base address per partition. During initialization, the available payload length register of all partitions are set to the maximum payload length and the base address registers are set to UINT64_MAX.

Stores destined to the same remote GPU are coalesced in an individual partition. When the first store arrives at a partition, its base address register is set to the address of the store right shifted by the number of address offset bits. The available payload length register is decremented by the sum of store length and the FinePack sub-header size. When a store arrives at a non-empty partition, the queue checks (1) if the address of the incoming store falls within the window of

$$[\text{base\_address}, \text{base\_address} + 2^{\text{num\_address\_offset\_bits}}),$$

and (2) the sum of the incoming store length and FinePack sub-header size will not exceed the value of the available payload length register. If these two conditions are not satisfied, the current contents of the queue partition are flushed and passed to the packetizer and the incoming store becomes the first store of the queue partition. On the other hand, if the two conditions

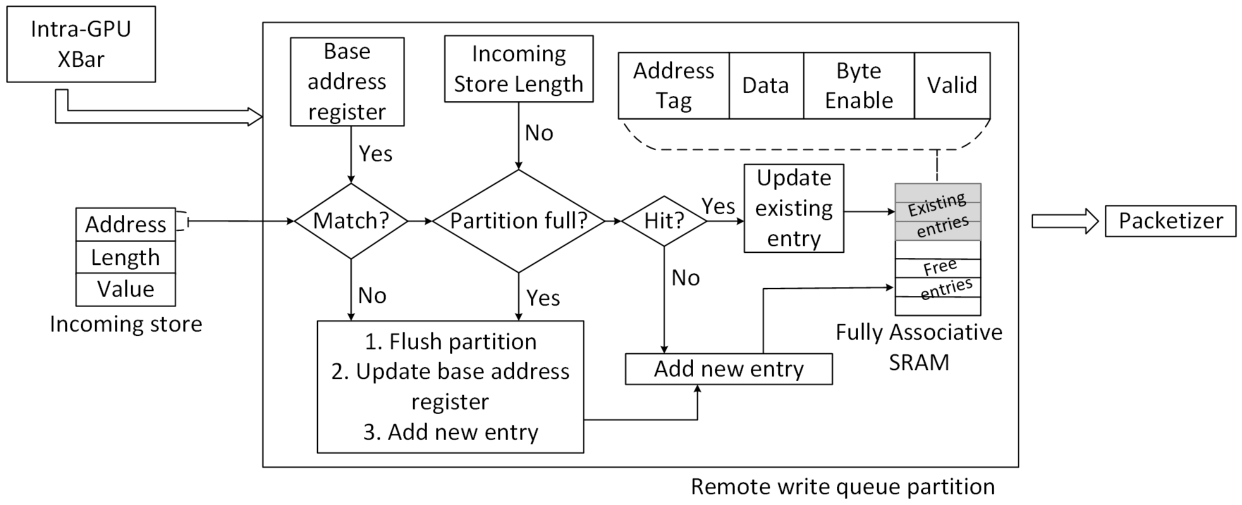


Fig. 8: FinePack remote write queue partition design. Every GPU will have one such partition per destination GPU.

are met, the queue partition performs an associative lookup to check if the incoming store address matches the value of an existing address tag. If there is a match indicating a queue hit, then the byte mask of the incoming store is ORed with the existing bytemask of the queue entry. The payload length register is updated to reflect the change. The incoming store will also overwrite the valid data bits of the corresponding queue entry. In case of a miss, the store creates a new entry in the queue partition, updates the byte mask and the payload length register. This matching logic can be realized through an FSM in hardware.

To ensure that the remote write queue partitions buffer sufficient data to target reaching the PCIe maximum payload size of 4kB, they are sized to 64 entries of 128B each which consumes 48kB total storage 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. For instance, in GV100 the total cache size (L1 + L2) is 16MB. 48kB required by the remote write queue is just 0.3% of the total cache capacity.

The entire 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 with respect 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. A GPU release operation requires the hardware implementation to flush in-flight stores to the point at which they become visible to all threads participating 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's memory consistency model. Note that when latency or burstiness of inter-GPU interconnect traffic constrains performance, the queue can be flushed after an inactivity timeout. However, we chose not to implement such timeouts to maximize the coalescing window and because flushing the queue when it becomes full was sufficient to maintain good link utilization.

To ensure same-address load-store ordering is maintained, a load operation to remote memory with an address that matches a write in the packetizer or remote write queue 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 and local loads.

Packetizer: The packetizer receives the entries flushed from a remote write queue partition and converts them into sub-packets in the FinePack format described in Section IV-A. Each individual remote write queue entry may need to be split 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 FinePack transaction embedded into PCIe and transferred to the interconnect.

De-packetizer: When a FinePack packet reaches the destination GPU, the de-packetizer disaggregates the stores into individual memory transactions before forwarding them into the GPU's memory system. The de-packetizer modifies the address field of each disaggregated packet by adding the offset from the sub-transaction header to the base address in the PCIe header. The de-packetizer must also include a 64 entry buffer of 128B each, because the deaggregated transactions cannot typically be consumed in the same cycle by L2.

No other changes are needed to the GPU architecture.

C. Discussion

Effect of Small Accesses on Local Memory Bandwidth: Although FinePack ensures reduced overheads on the inter-GPU interconnect, the communication to and from the

GPU's local memory within the GPU still occurs as sub-cacheline 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 inter-GPU interconnect.

Applicability to Read Accesses and Atomics: FinePack focuses on improving interconnect efficiency for applications that adopt a peer-to-peer store programming model for communication. In those applications, proactive data transfers via stores ensures that subsequent loads complete locally without having to traverse the interconnect on the critical path. Because on-demand loads stall the GPU threads issuing them, performing FinePack optimizations for loads could exacerbate their latencies and harm application performance. Hence, FinePack performs aggressive packing and coalescing only for stores. When remote atomic operations are issued, they are not coalesced and instead flush the previous entry with the same address in the remote write queue. However, atomic coalescing hardware has been previously proposed [9] and could be explored in the future.

Base Address Alignment: As described in Section IV-B, FinePack sets the base address of the remote write queue to the address of the first incoming store, with the low-order bits corresponding to the FinePack sub-transaction addressable range (Table II) masked off. This mechanism is simple, but may not be optimal if a data structure straddles an alignment boundary matching the addressable range. There are multiple ways to solve this. One approach is to use a more sophisticated remote write queue design that dynamically calculates the address range spanned by all currently queued stores. 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. The simplest approach is to set the base address using the upper bits of the address of the first store arriving at a partition. We evaluated the simplest approach to avoid unnecessary complexity and because the issues described here did not arise as first-order concerns in practice.

Partitioning of the Remote Write Queue: The remote write queue is partitioned to coalesce stores with the same destination. This can be done in multiple ways. In our evaluation, we set the number of partitions as the number of destination GPUs and designed the queue partitions to hold 4KB data addressed at 128B granularity (the maximum PCIe payload size). However, the size could be adjusted as performance needs dictate. If in larger systems it becomes too expensive to allocate the required storage, FinePack can scale down gracefully as the number of remote write queue entries dedicated to each destination GPU shrinks. 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 fully dynamic allocation, rather than partitioning the capacity in advance. We leave further exploration of such fine-tuning for future work.

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	144 bytes
Base address register	8 bytes
PCIe maximum packet size	4096 bytes
FinePack subheader size	5 bytes
FinePack subheader address offset	30 bits

TABLE III: Simulation parameters, based on NVIDIA GV100.

Compatibility with Memory Ordering Rules: Although the FinePack write queue reorders store operations, the GPU's weak memory model permits accesses to non-overlapping addresses to be reordered freely (assuming there is no synchronization in between). PCIe requires store TLPs to remain in order, so same-address ordering is maintained throughout. All other buffers and caches are unaffected by FinePack.

Applicability Beyond PCIe: Without loss of generality, we will focus on PCIe for our evaluation, though the techniques we have described will generalize to other common protocols such as NVLink or CXL. CXL for example reuses and extends PCIe, and thus FinePack is directly applicable. NVLink however uses byte enable fields for the entire payload and would likely would require slightly different encodings of the FinePack payload within the outer transaction. While a detailed specification for all inter-GPU protocols is beyond the scope of this work, we note that the small packet efficiency of PCIe and NVLink is similar for sub-cache line stores and the general approach of compressing multiple small stores into a single larger payload within an outer transaction should achieve similar benefits.

V. Experimental Methodology

We evaluate FinePack by extending the NVIDIA Architectural Level Simulator (NVAS) [47] comprising a system of four 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 [48] and replay these traces in the simulator. These traces contain CUDA API events, GPU kernel instructions, and memory accesses. The simulator models execution timing when replaying these traces. All timing aspects including remote store timings and subsequent dependencies are modeled within the simulator. To simulate the PCIe interconnect, we model the protocol packetization overheads and interconnect bandwidth numbers from the available PCIe specifications.

Our FinePack implementation in NVAS is configured with the parameters listed in Table III and evaluated on a suite of multi-GPU applications described below:

Jacobi: The Jacobi solver is an iterative algorithm used widely in numerical analysis to solve strictly diagonally dominant systems of linear equations [51]. The algorithm solves linear equations of the form $Ax = b$, where A is the coefficient matrix, b is an input vector, and x is the solution vector. For our evaluations we chose synthetically generated banded matrices which arise widely in finite element analysis as the coefficient matrix. On multi-GPU systems, the boundary values are shared with the neighbors; the communication pattern is peer-to-peer.

Pagerank: Pagerank algorithm is used for ranking webpages by assigning a 'Pagerank score'. Our implementation computes pagerank as an iterative, synchronous series of matrix-vector operations [52]. We evaluate pagerank on the Cage matrix [13]. Pagerank is an irregular application whose communication patterns vary based on the input dataset. For the chosen dataset, the communication pattern is peer-to-peer.

SSSP: Single Source Shortest Path (SSSP) algorithm computes the shortest path from a chosen location to the remaining locations in a dataset. We implement the Bellman-Ford variant of SSSP [53] on the indochina dataset [13]. SSSP is also an irregular application whose communication pattern is input dependent. For the indochina dataset we used for evaluation, the predominant communication pattern is many-to-many.

ALS: Alternating Least Squares (ALS) is widely used in recommender systems to iteratively factorize a rating matrix into a user matrix and an item matrix. Every iteration consists of two sub-iterations where the algorithm fixes the user or item matrix and optimizes the other [45]. We perform ALS on the rgg dataset [13], and the communication pattern is all-to-all.

CT: Our CT benchmark performs Model Based Iterative Reconstruction (MBIR), a computational technique used widely for low-dose CT scans. The algorithm we study is similar to that used in the FDA-approved GE Veo CT system and communicates with peer GPUs via all-to-all transfers [27].

EQWP: We also evaluate EQWP, Diffusion and HIT from the Tartan benchmark suite [23] because their strong scaling performance is bound by inter-GPU communication. EQWP [24] is a 3D Earthquake Wave Propagation model simulation and is based on 4-order finite difference method. In each iteration, each GPU exchanges halo regions with its neighbors and the communication pattern is peer-to-peer. We modify the application to replace MPI-based communication with the different communication paradigms we evaluate.

Diffusion: Diffusion [14], the second application we evaluate from Tartan benchmark suite [23] simulates the Heat Equation and Inviscid Burger's equation. Similar to EQWP, each GPU performs halo exchange with neighbors in each iteration via peer-to-peer transfers and we modify the code base to only replace MPI based communication with the paradigms under study.

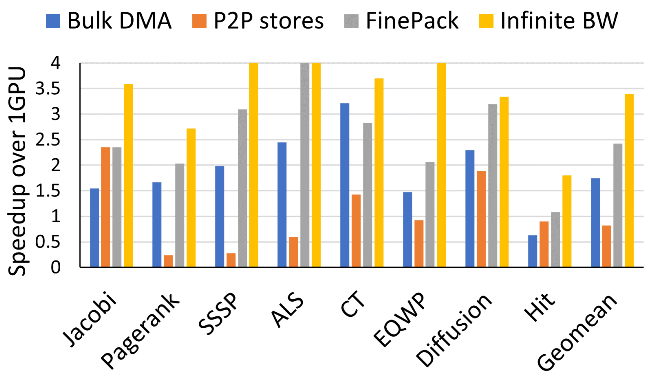


Fig. 9: 4-GPU speedups over a single GPU for different paradigms.

Hit: HIT [26], the final application from the Tartan suite computes Homogeneous Isotropic Turbulence as a series of FFT operations. It partitions the dataset along X-axis, transposes the coefficient matrix before/after FFT and communicates via all-to-all transfers.

For each multi-GPU application we evaluate two functionally equivalent implementations that maintain the same data and work sharing characteristics across GPUs, but use the memcpy and peer-to-peer store paradigms, described in Section III. FinePack works transparently, optimizing the peer-to-peer store paradigm to provide a more efficient bandwidth utilization. To establish an upper bound for the potential benefit of FinePack, we also compare it with a system with infinite inter-GPU bandwidth. We modeled this system by analytically eliding the data transfer time during an application's execution when using the memcpy paradigm.

VI. Results

We now examine the performance of FinePack on a multi-GPU system as described in Section V.

A. *Performance Results and Characterizations*

Figure 9 shows the speedup of using various multi-GPU communication paradigms on a four GPU system normalized to the performance of a single GPU baseline. The infinite bandwidth bar shows the total opportunity available via optimizing inter-GPU communication and on average it is 3.4×, indicating the highly parallel nature of these workloads. Peer-to-peer stores achieve considerable speedups in regular applications (Jacobi, diffusion) where the store granularity is 128B but performs poorly for others resulting in a net slowdown over a single GPU. Bulk DMA achieves a 2.1× performance improvement over peer-to-peer stores due to lower protocol overheads but is still 1.4× slower than FinePack.

Figure 10 provides a breakdown of the interconnect efficiency of the three studied inter-GPU communication schemes. As the figure shows, the bulk DMA paradigm results in negligible protocol overheads due to the coarse granularity of transfers. Despite low protocol overheads, the memcpy

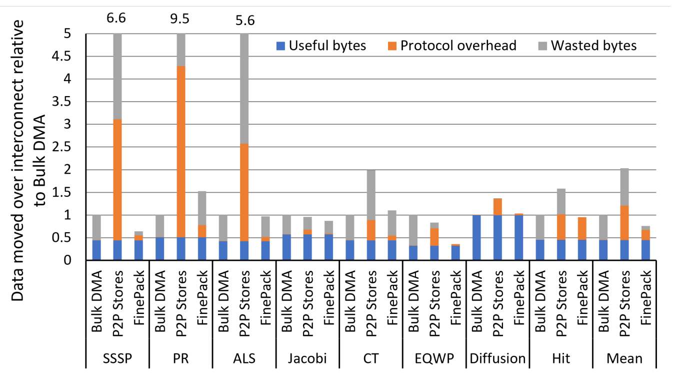


Fig. 10: Breakdown of total bytes transferred over the interconnect, normalized to inter-GPU bulk DMA. These bytes are categorized into 'Useful bytes', which are read by the destination GPU, 'Protocol overhead' which is the number of bytes required to perform all transfers, and 'Wasted bytes', which is bytes transferred that are never read or are overwritten by the source GPU.

paradigm suffers due to (1) its inability to overlap compute and communication, and (2) transfer of data that was not updated by the source and/or not accessed by the destination results in high percentage of wasted bytes. This leads to performance degradation compared to FinePack as shown Figure 9 and it is difficult to eliminate due to the coarse-grained nature of this paradigm.

P2P stores achieve a good overlap of compute and communication, but also exhibit poor performance as shown in Figure 9 due to the large 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. Figure 10 shows that 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, transfers 2.7× less data than peer-to-peer stores and 1.3× less data than bulk DMA as shown in Figure 10. This savings along with FinePack's ability to efficiently overlap compute and communication results in a mean performance improvement of 1.4× over bulk DMA and 3× over peer-to-peer stores as shown in Figure 9. We also note that FinePack results in a 24% reduction of data on the wire versus write combining alone.

Figure 11 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 42 stores on average into a single transaction before sending it to the interconnect, thus leading to the protocol overhead reduction shown in Figure 10. CT is a notable outlier from the trend and is able to pack fewer stores on average because the individual stores exhibit minimal spatial locality and thus FinePack does not outperform bulk DMA. Nevertheless, it still achieves good scaling as shown in Figure 9, because the application is not severely bandwidth bound.

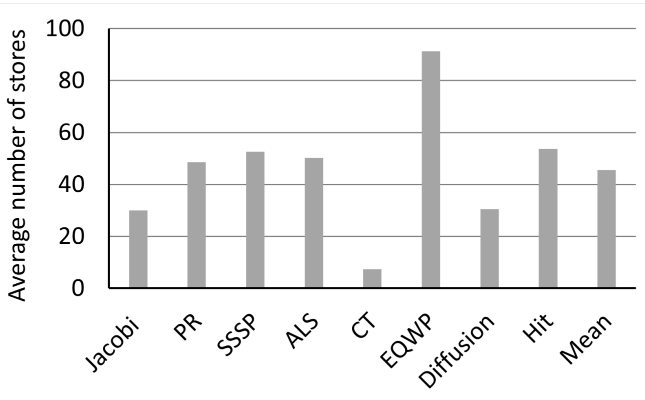


Fig. 11: Average number of stores aggregated in a single packet by FinePack.

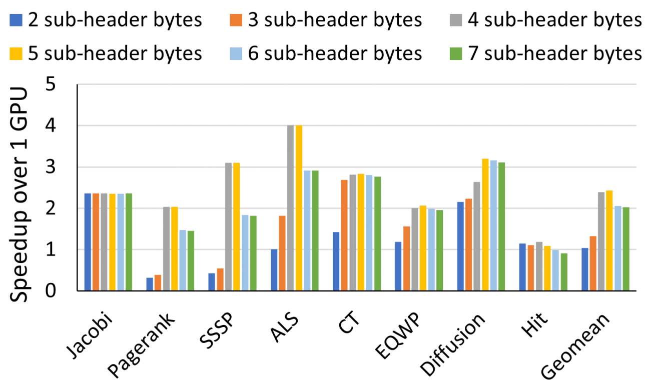


Fig. 12: FinePack performance sensitivity to variation of sub-header bytes.

Understanding Address Compression Tradeoffs: One of the important design decisions that has to be made when considering the FinePack optimization is how to sub-divide the store address into the address base and offset fields. As discussed in Section IV, 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 grows. Figure 12 shows that application performance grows upon increasing the sub-transaction header bytes and reaches the maximum at 4 sub-transaction header bytes, with virtually no change at 5 bytes. For our applications, 4 bytes (meaning 4MB coalescing regions) appears to be sufficient to capture majority of the data communication patterns within the workloads. However, beyond 5 bytes, performance begins to decrease because the protocol overhead grows, yet additional stores cannot be accommodated in a single packet because FinePack hit the maximum payload size limit. While there is variation among workloads, we observe that 4-5 sub-transaction header bytes works well across all applications. We choose to use 5 bytes in our evaluations.

Sensitivity to Interconnect Bandwidth: Figure 13 shows the geometric mean performance gains of FinePack over bulk DMA and peer-to-peer stores when increasing the inter-GPU bandwidth, while leaving GPU's compute capabilities constant. We observe that bulk DMA and peer-to-peer store designs improve performance significantly with each step in bandwidth. However, at no step (until bandwidth is unlimited) do they achieve the performance of FinePack.

Though inter-GPU interconnect bandwidth will grow, we expect it to remain magnitudes lower than the local memory bandwidth even in future multi-GPU systems where HBM bandwidth is also growing each generation.

B. Discussion

Alternate FinePack Designs: As a part of this work, we performed opportunity studies evaluating an alternate design where protocol overheads are minimized by separating the base address and other common fields into a special PCIe *configuration packet*. This packet defines the characteristics for store-packets that follow it, until the next configuration packet is received, 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.

Our analytical modeling showed that the protocol overhead reduction is not as efficient as FinePack when using this alternative. FinePack packs multiple stores as sub-transactions within a single PCIe packet, thus needing just one common sequence number, ECRC, and LCRC per packet. In a stateful config-packet design, though the common header fields across multiple stores are sent as a single configuration packet, the stores that succeed remain independent PCIe packets, requiring their own sequence number and CRC fields, which produce an additional 10-byte overhead per store compared to FinePack. For a packet containing 32-64 stores (FinePack typically coalesces 42 stores before emitting a packet), this alternate design is approximately 18% less efficient.

FinePack Overheads: Because FinePack overheads are incurred only on the store side, for an application with P2P store programming model, they do not fall on the critical path of compute execution. Our simulator model includes all the overheads incurred by FinePack. The cost to flush a remote write queue at the synchronization barriers will be dwarfed by the cost of the synchronization barrier itself. The area requirement for FinePack remote write queue is less than 0.05% of the area of existing caches in NVIDIA's recent GA100 GPU. Even on a 16 GPU system, the total remote write queue partition size is 120kB per GPU, which is dwarfed by the L2 size (40MB).

Comparison with Other Proactive GPU Transfer Systems: GPS [31] is a recent hardware-software co-designed 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 traffic to unused replicas. GPS also includes a write combining buffer, but because it performs coalescing at the cacheline granularity, it

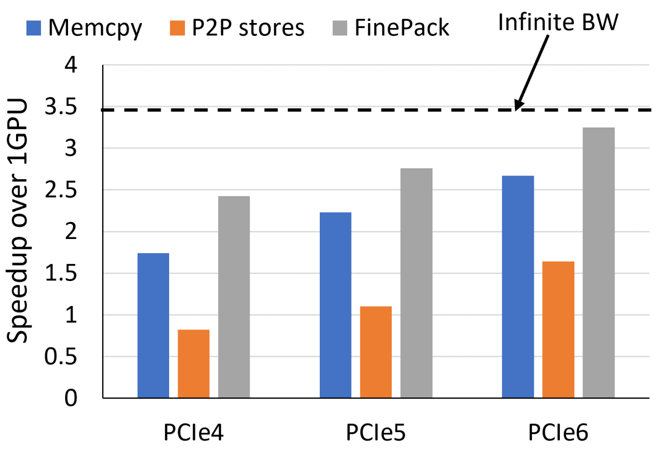


Fig. 13: Performance sensitivity to interconnect bandwidth. PCIe6 bandwidth is similar the the highest performance NVLink interconnects available today.

cannot achieve good coalescing for highly divergent stores and still suffer from small store network inefficiency. In addition, it requires programmer effort to identify the profiling region and requires the programmer to port their application to new memory management APIs. While effective, it does require changes to the GPU's core virtual memory architecture to be implemented efficiently. We directly compared FinePack to GPS for individual applications that had the same reference implementation. We found 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 best. In other workloads where these unneeded transfers result in larger inefficiencies, FinePack performs better than GPS, while requiring no application porting to utilize a new GPU-only centralized memory subscription mechanism. On average, across our evaluated workloads we find FinePack is 17.8% slower than GPS on a 4-GPU PCIe system, however FinePack is a significantly simpler modification to the GPU's architecture and remains fully transparent to the programmer without need for new special libraries or API usage. We note that GPS and FinePack are not mutually exclusive; they could serve as complementary optimizations in future systems.

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 MBs 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. On a 16-GPU system connected via a projected PCIe 6.0 interconnect, FinePack outperforms peer-to-peer stores by $3\times$ and bulk DMA by $1.9\times$, without affecting the programming model, GPU runtime, or inflicting invasive changes on the GPU memory system.

VII. Related work

Small message aggregation: Small message aggregation to improve network efficiency has been proposed both at the software and hardware level. There are a variety of software approaches, ranging from runtimes [30], [49], to compilers [2], or application level optimization [7]. 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 particularly successful use case of software based network aggregation is the acceleration of collective communication [19]. Gravel [36] performs message aggregation in an external CPU managed queue and is intended for Ethernet/infiniband networks at large communication sizes, where as FinePack performs coalescing and address compression within the GPU before injecting the data into the network.

Network level approaches to aggregation [8], [18] are designed as part of the network routers and aggregate messages as they travel within the network. They are transparent to the user and application while providing benefit because the routers can inexpensively monitor the effectiveness of the aggregation and use it as needed. Our work proposes an application transparent, endpoint based approach that is also transparent to the network itself, as it logically sits at the interface between the GPU and the network ports. Our approach is based on the aggregation of memory references following a scheme that is reminiscent of that found inside compressing TLBs [29], [42], [54] as a way improve virtual address space coverage while requiring a smaller number of TLB entries.

Multi-GPU performance: PROACT [32] is a joint compile and runtime system that intelligently orchestrates shared data in multi-GPU systems via proactive transfers. It fine-tunes inter-GPU data movement for the application's needs, also 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-writes to integrate its framework into each application. FinePack on the other hand, is completely programmer transparent and offers significant performance gains without the need for profile driven optimization.

Prior work [3], [4], [6], [20], [28], [55] explores other mechanisms both at the HW and SW levels to improve the performance in multi-GPU systems. Our work adds to this by exploring optimizing small access efficiency to enable strong scaling in multi-GPU systems. Some work [40], [41], [44] has also proposed techniques for inter-GPU coherence. FinePack avoids an expensive coherence protocol while also maintaining GPU memory model compatibility. Several works propose multi-GPU memory management solutions. Griffin [4] optimizes page migration to improve multi-GPU performance and CARVE [55] caches remote data in local DRAM to improve locality. Prior work [46] explores the existence of common data patterns and the use of memory compression algorithms to save bandwidth and hence improve performance in multi-GPU systems. It is orthogonal and complementary to FinePack.

NUMA memory management: Dashti et al. [12] develop a Linux memory placement algorithm for mitigating address traffic congestion in NUMA systems and there is significant prior work [1], [15], [21], [39], [56] performing NUMA-aware optimizations for CPU based systems. Others have also explored techniques that require new hardware-based peer caching for GPUs [5], [11], [25], [38], [43].

VIII. Conclusion

Performing fine-grained writes to other GPUs' 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. 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 over the inter-GPU interconnect. By leveraging the locality inherent in the physical address stream egressing a single GPU, bound for a single destination GPU, FinePack is able to achieve a $2.7\times$ efficiency improvement over peer-to-peer stores that translates into an average program speedup of $3\times$. We believe techniques such as FinePack that rely on hardware innovations while remaining transparent to the application developer (to ease development overhead) are essential for realizing scalable multi-GPU systems in the future.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Center for Applications Driving Architectures (ADA), one of the six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

References

[1] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. 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] M. Alvanos, M. Farreras, E. Tiotto, and X. Martorell, "Automatic Communication Coalescing for Irregular Computations in UPC Language," in *Conference of the Center for Advanced Studies on Collaborative Research(CASCON)*, 2012, pp. 220–234.

[3] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *International Symposium of Computer Architecture (ISCA)*, 2017.

[4] T. Baruah, Y. Sun, A. Dinçer, M. S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[5] A. Basu, S. Puthoor, S. Che, and B. M. Beckmann, "Software Assisted Hardware Cache Coherence for Heterogeneous Processors," in *International Symposium on Memory Systems (ISMM)*, 2016.

[6] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: Asynchronous Multi-GPU Programming Model with Applications to Large-scale Graph Processing," *Transactions on Parallel Computing (TOPC)*, vol. 7, no. 3, pp. 1–27, 2020.

[7] V. G. Castellana, M. Drocco, J. Feo, J. Firoz, T. Kanewala, A. Lumsdaine, J. Manzano, A. Marquez, M. Minutoli, J. Suetterlein *et al.*, "A Parallel Graph Environment for Real-World Data Analytics Workflows," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[8] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow *et al.*, "Looking Under the Hood of the IBM Blue Gene/Q Network," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[9] P. Dalmia, R. Mahapatra, and M. D. Sinclair, "Only Buffer When You Need To: Reducing On-Chip GPU Traffic with Reconfigurable Local Atomic Buffers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 676–691.

[10] J. Danskin and D. Foley, "Pascal GPU with NVLink," in *IEEE Hot Chips 28 Symposium (HCS)*, 2016, pp. 1–24.

[11] M. Dashti and A. Fedorova, "Analyzing Memory Management Methods on Integrated CPU-GPU Systems," in *International Symposium on Memory Management (ISMM)*, 2017.

[12] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[13] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[14] M. Diaz, "Multi-GPU (CUDA-MPI) Baseline Implementation of Heat Equation and the Inviscid Burgers' Equation," https://github.com/wme7/MultiGPU_AdvectionDiffusion, last accessed on 12/05/2022.

[15] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages," *Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 1–25, 2014.

[16] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[17] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," *arXiv preprint arXiv:1807.05358*, 2018.

[18] N. Jiang, L. Dennison, and W. J. Dally, "Network Endpoint Congestion Control for Fine-Grained Communication," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[19] L. V. Kalé, S. Kumar, and K. Varadarajan, "A Framework for Collective Personalized Communication," in *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2003, pp. 9–pp.

[20] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-Aware Unified Memory Management in GPUs for Irregular Workloads," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[21] B. Lepers, V. Quéma, and A. Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," in *USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[22] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 31, no. 1, pp. 94–110, 2019.

[23] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite," in *International Symposium on Workload Characterization (IISWC)*, 2018.

[24] Z. Liu, "Efficient Large-scale Parallel Stencil Computation on Multi-Core and Multi-GPU Accelerated Clusters," https://github.com/lzhengchun/b2r, last accessed on 12/05/2022.

[25] D. Lustig and M. Martonosi, "Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[26] M. Martın, "HIT: A Parallel GPGPU Code to Simulate Homogeneous Isotropic Turbulence," https://github.com/albertovelam/HIT_MPI, last accessed on 12/05/2022.

[27] M. G. McGaffin and J. A. Fessler, "Alternating Dual Updates Algorithm for X-ray CT Reconstruction on the GPU," *IEEE Transactions on Computational Imaging*, vol. 1, no. 3, pp. 186–199, 2015.

[28] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the Socket: NUMA-aware GPUs," in *International Symposium on Microarchitecture (MICRO)*, 2017.

[29] S. Mittal, "A Survey of Techniques for Architecting TLBs," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, p. e4061, 2017.

[30] A. Morari, O. Villa, A. Tumeo, D. Chavarria Miranda, and M. Valero Cortés, "GMT: Enabling Easy Development and Efficient Execution of Irregular Applications on Commodity Clusters," in *International Conference for High Performance Computing, Networking, Storage and Analysis, (SC)*, 2013.

[31] H. Muthukrishnan, D. Lustig, D. Nellans, and T. Wenisch, "GPS: A Global Publish-Subscribe Model for Multi-GPU Memory Management," in *International Symposium on Microarchitecture (MICRO)*, 2021.

[32] H. Muthukrishnan, D. Nellans, D. Lustig, J. Fessler, and T. 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.

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

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

[35] 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.

[36] M. S. Orr, S. Che, B. M. Beckmann, M. Oskin, S. K. Reinhardt, and D. A. Wood, "Gravel: Fine-Grain GPU-Initiated Network Messages," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[37] C. Pham, "Comparison of Message Aggregation Strategies for Parallel Simulations on a High Performance Cluster," in *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2000.

[38] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[39] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement," in *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[40] X. Ren and M. Lis, "Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[41] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[42] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," in *International Symposium on Computer Architecture (ISCA)*, 1995.

[43] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[44] A. Tabbakh, X. Qian, and M. Annavaram, "G-TSC: Timestamp Based Coherence for GPUs," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[45] G. Takács and D. Tikk, "Alternating Least Squares for Personalized Ranking," in *ACM Conference on Recommender Systems*, 2012, pp. 83–90.

[46] M. K. Tavana, Y. Sun, N. B. Agostini, and D. Kaeli, "Exploiting Adaptive Data Compression to Improve Performance and Energy-Efficiency of Compute Workloads in Multi-GPU Systems," in *International Parallel*

and Distributed Processing Symposium (IPDPS). IEEE, 2019, pp. 664–674.

[47] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, N. Jiang, and D. Nellans, "Need for Speed: Experiences Building a Trustworthy System Level GPU Simulator," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[48] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[49] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "TRAM: Optimizing Fine-Grained Communication with Topological Routing and Aggregation of Messages," in *International Conference on Parallel Processing (ICPP)*, 2014.

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

[51] Wikipedia, "Jacobi Method," 2018, https://en.wikipedia.org/wiki/Jacobi_method, last accessed on 02/12/2022.

[52] Wikipedia, "PageRank," 2018, https://en.wikipedia.org/wiki/PageRank last accessed on 02/12/2022.

[53] Wikipedia, "Bellman Ford Algorithm," 2022, https://en.wikipedia.org/wiki/Bellman\-Ford_algorithm, last accessed on 02/12/2022.

[54] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation Ranger: Operating System Support for Contiguity-Aware TLBs," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[55] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[56] K. Zhang, R. Chen, and H. Chen, "NUMA-Aware Graph-Structured Analytics," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.