DMA Collectives for Efficient ML Communication Offloads

Suchita Pati
Advanced Micro Devices, Inc.
United States
suchita.pati@amd.com

Mahzabeen Islam
Advanced Micro Devices, Inc.
United States
mahzabeen.islam@amd.com

Shaizeen Aga
Advanced Micro Devices, Inc.
United States
shaizeen.aga@amd.com

Mohamed Assem Ibrahim
Advanced Micro Devices, Inc.
United States
mohamed1.ibrahim@amd.com

Abstract

Offloading machine learning (ML) communication collectives to direct memory access (DMA) engines, available on most state-of-art commercial GPUs, has emerged as an interesting and low-cost solution to efficiently overlap computation and communication in inference and training. Doing so delivers superior concurrent performance by freeing up all GPU cores for computation and also lowers interference in the memory sub-system (caches). While DMA collectives show strong promise, prior works have only studied them in limited context (bandwidth-bound transfer sizes only, performance-only).

To address this, we provide a comprehensive performance, power/energy and synchronization costs analysis of offloading ML communication collectives (all-gather, all-to-all) to DMA engines on state-of-the-art AMD Instinct™ MI300X GPUs. Our analysis reveals that, compared to the state-of-the-art RCCL communication collectives library, DMA collectives are at-par or better for large sizes (10s of MB to GB) in terms of both performance (16% better) and power (32% better). However, they significantly lag for latency-bound small sizes; 4.5× and 2.5× slower for all-gather and all-to-all, respectively. We provide a detailed latency breakdown of a DMA transfer and identify that DMA command scheduling and synchronization costs can limit DMA collective performance. To tackle this, we harness existing DMA architecture innovations, hitherto untapped, to build optimized DMA collectives and demonstrate their efficacy on real hardware. Our optimized implementations considerably close the performance gap for DMA collectives at smaller sizes (30% slower and 20% faster all-gather and all-to-all, respectively) and further improves performance (by 7%) and power savings at larger sizes (3-10%). Overall, this work represents a significant step toward making DMA collectives suitable for adoption in mainstream collective libraries.

1 Introduction

Scaling of ML models and their training datasets have driven their efficacy and consequently have led to their widespread adoption across variety of domains. This in turn has led to ML training and inference to be often distributed over multiple GPUs thus necessitating focused optimization of inter-GPU communication costs. That is so as while slicing work across GPUs helps parallelize ML computations, any additional time added due to communication reduces the effective parallelism [26].

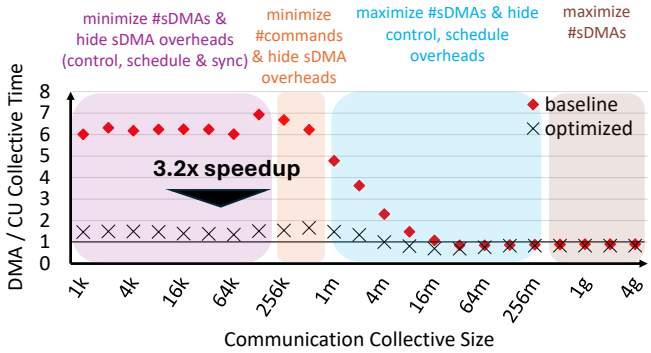

Figure 1. Bridging the performance gap between DMA and GPU compute-unit (CU)-based collectives from RCCL with unique optimizations across the size spectrum.

To better tackle this tradeoff, ML algorithmic advancements often aim to overlap ML communication (termed as collectives) with concurrent computation (e.g., fully sharded data parallelism, or FSDP, hides communication of one layer with computations of another/other layer(s) [32]). There have also been several efforts to improve communication and compute overlap through their fine-grained interleaving - communicating a tile of data as soon as the producer generates it [11, 14, 15, 17, 21, 30, 31, 33].

Further, to do such overlap efficiently, prior works [24] have looked at specialized communication accelerators so as to lower interference to concurrent computation. An interesting and low-cost solution in this regard is to harness existing direct memory access (DMA) engines, available on most state-of-art commercial GPUs, for ML collectives. Doing so has been demonstrated to deliver superior computation-communication concurrency by freeing up all GPU cores

for computation and also lowers interference in the memory sub-system (caches) [7].

While DMA collectives show strong promise, prior works have only studied them in limited context (bandwidth-bound transfer sizes only, performance-only). To address this, we extend existing know-how of DMA collectives considerably and provide the first comprehensive performance, power (energy) and synchronization costs analysis for collectives that harness existing DMA engines on the state-of-the-art AMD Instinct™ MI300X GPUs.

As shown in Figure 1, our analysis confirms the promise DMA collectives hold: for large size transfers (10sMB to GB), DMA collectives are at-par or deliver superior performance to state-of-the-art communication collectives RCCL library which harnesses GPU compute cores or units (CUs) for communication. Not only this, DMA collectives also deliver power (energy) savings for this size range (32%). While promising for such bandwidth-bound transfer sizes, our analysis shows that DMA collectives considerably lag behind existing collectives libraries for latency-bound small-size transfers (4.5× and 2.5× slower all-gather and all-to-all, respectively). Note that, this size range is particularly relevant for inference and is getting increasingly relevant as ML frameworks increasingly rely on fine-grain computation and communication overlap as discussed above.

To tease out the reasons for this performance gap, we carefully instrument GPU software stack to provide a detailed benchmarking of latency composition of a single DMA transfer. Since a ML collective is comprised of multiple DMA transfers, a detailed latency composition of a single DMA transfer provides useful insights into optimizations necessary for performant DMA collectives. This detailed benchmarking demonstrates that DMA command scheduling and synchronization times can be as high as 60% of the total transfer time.

To tackle this performance gap, we next identify interesting DMA architecture capabilities, hitherto untapped, that help address the command scheduling and synchronization costs associated with DMA transfers. Specifically, while existing DMA collectives harness only the vanilla *copy* DMA command which copies data from single source address to destination address, we harness two novel DMA commands (*broadcast* and *swap*). As these commands express multiple copy operations with single command, using them lowers the total number of DMA commands necessary for a ML collective and directly addresses DMA command scheduling overheads. Second, while commercial GPUs employ multiple DMA engines and using multiple engines for a single collective allows increased parallelism, it also leads to increased synchronization costs. We observe here that, we can harness the ability of DMA engines to efficiently overlap execution of series of DMA transfers (*back-to-back copy*) to lower DMA engines employed and thus lower synchronization costs. Finally, we also demonstrate the efficacy of *prelaunch*, that is, scheduling of DMA commands off the critical path in closing the performance gap of DMA collectives.

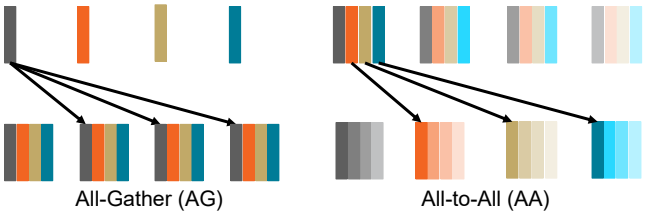

Figure 2. ML collectives (left) All-gather (right) All-to-All.

We build DMA collectives prototypes on real hardware which harness the above DMA architecture capabilities. Using our prototypes for all-gather and all-to-all collectives, we demonstrate that we can considerably close the performance gap between DMA and CU collectives (from 4.5× and 2.5× slower to only 30% slower and 20% faster all-gather and all-to-all, respectively) and deliver additional power savings as well (3-10%). Finally, we also study the implications and resultant synchronization costs of integrating DMA collectives in ML applications and study the impact of DMA collectives in presence of producer-consumer dependencies between DMA collectives and other GPU kernels. Our analysis demonstrates that DMA collectives can be successfully incorporated along with other GPU kernels.

Overall, the key contributions of this work are as follows:

- Given the promise shown by DMAs to enable efficient computation-communication overlap for ML, we provide a comprehensive performance, power (energy) and synchronization costs analysis for DMA collectives that harness existing DMA engines on GPUs.
- Our analysis demonstrates that DMA collectives are promising from both performance and power perspective at large transfer sizes (10s MB to GB) but considerably fall behind CU collectives for small transfer sizes (4.5× and 2.5× slower all-gather and all-to-all).
- Next, via careful instrumentation of GPU software stack, we identify that the reason for the above performance gap is high DMA command scheduling and synchronization times (up to 60% of total transfer time).
- To tackle this shortcoming of DMA collectives, we harness existing DMA architecture innovations, hitherto untapped, to build optimized DMA collectives for all-gather and all-to-all ML collectives.
- We demonstrate the efficacy of our optimized DMA collectives on real hardware (AMD Instinct™ MI300X GPUs) and our evaluation shows that they deliver 3.2× performance uplifts and 3-10% power savings.

Overall, we make significant strides in making DMA collectives mainstream and enabling efficient computation and communication overlap on commercial GPUs.

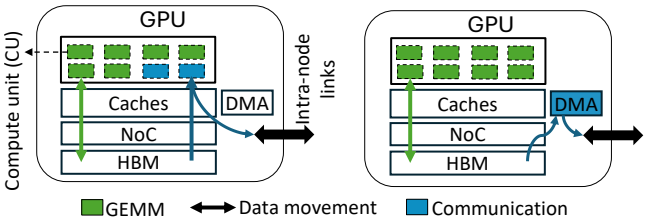

Figure 3. Concurrent Compute with Communication (left) using CUs (right) offloaded to sDMA engine.

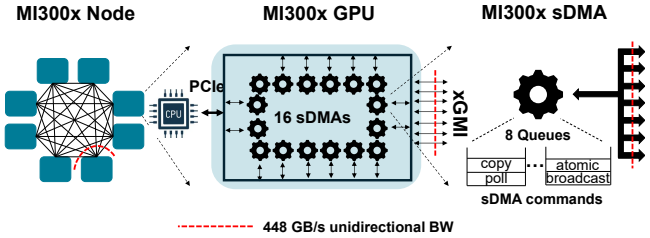

Figure 4. sDMA engines in AMD MI300X Infinity Platform.

2 ML Collectives and Need for Efficient Offload

2.1 All-gather and All-to-all ML Collectives

As model sizes and inputs scale, efficiently slicing weight (and/or activation) tensors across multiple GPUs and gathering them as needed is crucial for both performance (parallelism) and functionality (memory capacity). This is commonly used in fully sharded data parallelism (FSDP) [32] and sequence parallel (SP) [19]. Such aggregation of tensors is achieved by an *all-gather* (AG) collective shown in Figure 2(left), where each GPU starts with a sub-array and receives the remaining sub-arrays from other GPUs. In other words, each GPU sends its respective sub-array to all other GPUs. Another commonly employed collective is *all-to-all* (AA), wherein each GPU starts with a complete array but exchanges sub-arrays with all other GPUs as shown in Figure 2(right). In other words, all participating GPUs collectively perform a transpose operation. Mixture-of-expert (MoE) models in an expert-parallel setup use AA; each GPU exchanges tokens with other GPUs based on the tokens' expert preference as determined by the routing layer [22].

Models also heavily use *reduce-scatter* (RS) collective to reduce gradients from data-parallel model instances and to aggregate partial dot-products in tensor parallelism [26]. RS has a similar communication pattern as AA, except the sub-arrays received from other GPUs are reduced (e.g., summed) with the local sub-array. Since DMAs lack compute support, not all of RS can be offloaded to DMAs. Thus, we focus in this work on optimizing DMA offload of AG and AA collectives and discuss possible optimizations for RS in Section 6.1.

2.2 Concurrent Compute and Collectives

To reduce communication overheads in distributed ML setups and ensure high scaling efficiency, the collectives described in Section 2.1 are often executed concurrently with compute operations [13, 23, 32]. For instance, FSDP overlaps the compute operation of one model layer with the AG of weights from the next layer(s) in a coarse-grained manner [32]. Similarly, Deepseek MoE overlaps compute and communication across multiple independent instances [13]. In contrast, other mechanisms, such as SP and single-instance MoE, lack independent compute-communication operations preventing such coarse-grained overlaps. However, several studies have proposed fine-grained overlap techniques, where tiles or blocks of data generated by a producer kernel are immediately communicated, rather than waiting for the kernel to finish. Overall, overlapping compute and communication is crucial for efficiently scaling ML models.

2.3 Resource Contention and Need for Offload

Common GPU implementations of these collectives from state-of-the-art libraries such as RCCL [9, 12, 20] use GPU compute units (CUs) to orchestrate the communication. While these implementations are highly optimized for the collectives' isolated execution, they stand to achieve less than ideal performance when overlapped with compute operations [7, 24]. This is due to interference in both compute and memory subsystem resources as shown in Figure 3(left), which slow down compute and/or collective operations. Offloading communication to other accelerators can help free up compute and cache/memory resources for compute operations and reduce this contention [7, 24]. Prior work offload communication which reduces its GPU compute and memory requirements but propose a dedicated hardware accelerator for it [24]. In this work, we focus on leveraging *existing* DMA engines, available on most state-of-art commercial GPUs, to offload these communication collectives and alleviate CU and cache contention as shown in Figure 3(right).

2.4 DMAs in AMD Instinct™ MI300x

State-of-the-art GPU systems have considerable DMA capabilities. As shown in Figure 4, an AMD MI300X Infinity Platform has eight AMD Instinct™ MI300X GPUs that ML collectives are deployed across. Each GPU is fully connected with all other GPUs using AMD Infinity Fabric™ links, each of which has a bi-directional bandwidth (BW) of 128 GB/s (64GB/s unidirectional). Thus, each GPU can communicate with all other GPUs with a total BW of 448GB/s.

Each GPU has eight accelerator complex dies (XCD) [28], which are comprised of CUs. The XCDs are vertically stacked over four I/O dies (IOD) [28, 29]. The IODs are comprised of AMD Infinity Cache™, the memory interface to the on-package HBM as well as 16 system-DMA (sDMA) engines. As shown in Figure 4, each sDMA has access to all PCIe Gen5 and xGMI links for both CPU-GPU and GPU-GPU communication. Therefore, each sDMA can communicate with

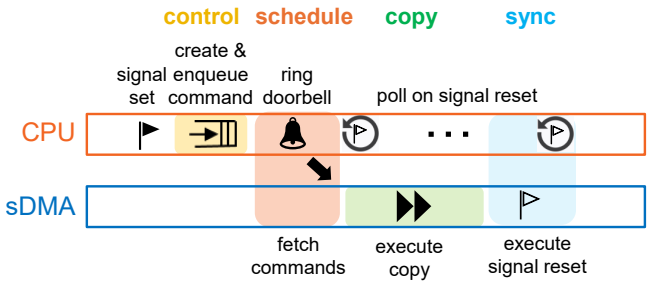

Figure 5. Phases of an sDMA copy offload.

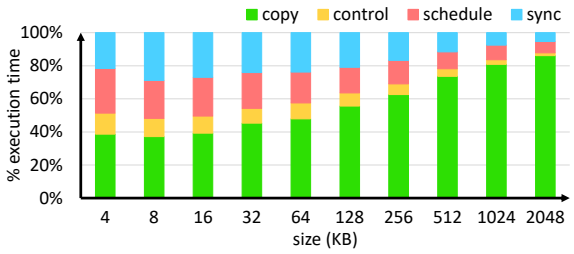

Figure 6. Latency breakdown of an sDMA copy.

any other GPU in the node. A host (usually CPU) enqueues commands to DMA queues (detailed in Section 3.2). sDMAs support several commands [1, 8], some key ones include *copy* to copy data from any source memory to any destination memory location, *broadcast* command to copy data from one source to two destinations, a *swap* command to swap data at any two memory locations, an *atomic* command which supports several 32b/64b atomic opcodes (used for signaling), and a *poll* command that polls on a memory location for an expected value. We discuss how these commands can help design ML collectives in Section 4.

2.5 Need for Efficient Offload

While sDMAs have been shown to deliver superior computation-communication concurrency by freeing up all GPU cores and limiting memory sub-system interference (caches), they have been studied only in limited contexts such as large bandwidth-bound transfer sizes. And furthermore, the focus has largely been on their performance benefits and not power/energy [7].

Our analysis confirms DMA collectives' benefits at larger sizes (10sMB to GB), where they are at-par or achieve superior performance to CU-based RCCL collectives. However, we find that DMA collectives considerably lag behind CU-based collectives for latency-bound small-size transfers; for sizes <32MB, they are on average 4.5× and 2.5× slower (maximum 7× slower) for AG and AA, respectively. We show this with the `baseline` (or `pcpy` detailed in Section 4.2.1) performance in Figure 1 for AG. This size range is crucial for ML inference (e.g., LLM decode) and has also become increasingly relevant as ML frameworks employ fine-grain computation and communication overlap, requiring communication at the granularity of small blocks/tiles generated by kernels.

Thus optimizing DMA collectives for these sizes such that they are at-par or better performing that their CU-counterparts is crucial for their efficient offload and overlap.

3 DMA Offload Phases: Where Goes Time?

Narrowing the performance gap between CU and DMA-based collectives at smaller sizes requires understanding where time is spent with DMA offload. Therefore, as a first step, we study the distinct phases of a single peer-to-peer DMA copy, which is the smallest unit of work applications can offload to DMAs with HIP/HSA API calls.

3.1 DMA Offload Benchmarking Methodology

Since most of the DMA functionality is within AMD HIP runtime (ROCr) [8] which handles several other tasks, we create a microbenchmark emulating only its DMA offload behavior (details in Section 3.2). As in ROCr, we use ROCt library which provides user-level APIs to interact with the GPU driver. This gives us low-level control on DMA functioning such as using its timing capabilities (timestamp command) to measure individual command latencies and build collectives described in Sections 4.2.

3.2 Phases of a DMA Offload

Figure 5 illustrates phases of a sDMA offload for a peer-to-peer copy, which is a fundamental primitive that we use to implement most ML collectives. Like GPU's compute workload / kernels, sDMA workload is also controlled by the CPU. Therefore, when a user/application requests a sDMA transfer from one (source) GPU to another GPU (destination), the CPU creates and enqueues commands to a sDMA's (of either the source or destination GPU) ring buffer or queue which resides in the system memory. We term this first step as the *control* phase. The CPU then notifies the sDMA of the work by ringing the doorbell which entails updating the doorbell pointer to point to its write pointer on the queue. A change in doorbell pointer wakes the sDMA controller which then fetches commands from the ring buffer into its internal buffers for processing. We call this CPU to sDMA handover as the *schedule* phase. The sDMA then executes the peer-to-peer copy command, which we call the *copy* phase; it decodes, performs address translations, and generates reads/writes from/to HBM memory of source/destination GPU, respectively, for the copy. Finally, sDMAs use signals to synchronize with the CPU, notifying it of the copy's completion. Therefore, a copy is usually followed by a signal update or sync sDMA command (a system memory write or atomic) which the CPU polls on. We term this signal command execution / CPU notification as the *sync* phase.

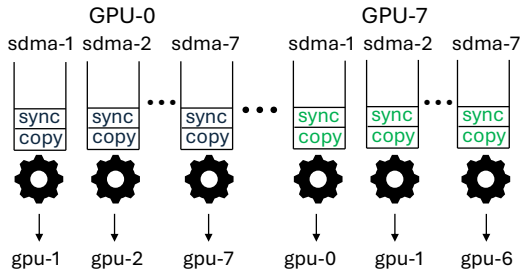

Figure 7. Parallel-copy based collective (pcpy).

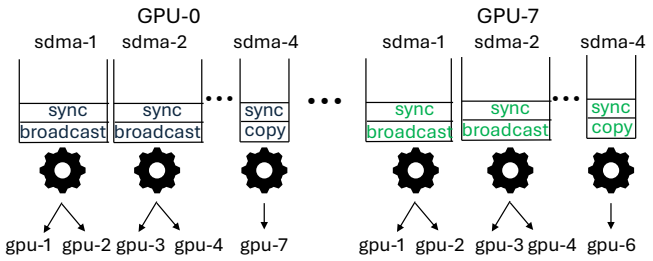

Figure 8. Broadcast-based allgather (bcst).

3.3 DMA Offload Latency Breakdown

Figure 6 shows the latency breakdown of a single peer-to-peer copy operation between two GPUs for a range of sizes (4KB to 2MB). Intuitively, since the cost of the copy phase, i.e., cost of local/remote reads/write, increases with transfer size while the latencies of other phases remain constant, the proportion of time spent on copy phase increases with size. Furthermore, across all sizes, the phases can be ordered as copy > schedule ~ sync >> control in terms of their runtime contribution. However, the schedule and sync latencies are a considerable portion of sDMA copy time. Overall, the non-copy phases - control, schedule and sync - account for ~60% of the time at the smallest of sizes and are small (<20%) only when copy sizes are >1MB.

ML collectives require multiple (i.e., $n * (n-1)$ where n is the number of devices) independent peer-to-peer transfers (Figure 2). Offloading such ML collectives thus requires the host to create several commands, schedule them on queue(s)/sDMA(s) and synchronize with them. These considerably scale the control, schedule and sync phases and limit DMA collective performance for small-to-medium size as shown by `baseline` in Figure 1. Thus, optimizing sDMA collectives to minimize the non-copy phase time is key to achieving superior performance at KB-MB sizes.

4 DMA Collective Design

As discussed in Section 3, communication using DMA introduces overheads that can diminish the benefits of offloading from compute units (CUs). These overheads are further amplified in the context of ML collectives, where a GPU must communicate with all other GPUs. In this section, we first identify the factors that contribute to these costs, and discuss strategies that can help mitigate and/or hide these costs. We then use these to create optimized implementations for allgather and alltoall DMA-based collectives.

4.1 Factors Impacting DMA Offload Costs

4.1.1 Count and Type of DMA Commands. As shown in Figures 5 and 6, control and schedule phases considerably affect execution times, contributing up to 10-40% of the time for transfers up to 1MB. Control phase involves the host creating and enqueueing commands into sDMA queues in system memory, while schedule phase involve fetching commands from these queues to the sDMA engine. Thus, to minimize control and schedule time in ML collectives which require many-to-many GPU communication, it is essential to limit the number of sDMA commands issued.

sDMA engines in a AMD Instinct™ MI300x support two types of commands that combine the functionality of two copy commands. The *broadcast* command specifies a single source and two destination memory locations, allowing the sDMA engine to perform a single read from the source and write to two distinct memory locations, potentially across different GPUs. This not only reduces redundant memory reads, as discussed in Section 6.2, but also combines the effects of two copy commands, thus reducing the control time. Similarly, the *swap* command exchanges data between two memory addresses with a single command, replacing what would otherwise require two copies. This not only cuts control costs in half but also performs the data exchange in-place, avoiding the need for a temporary buffer to prevent data overwriting and thereby saving memory capacity.

4.1.2 Number of DMA Engines. Engaging multiple sDMA engines per GPU (Section 2.4), each handling a copy packet, can potentially parallelize transfers from a given GPU. However, scheduling each operation to the individual sDMA engines requires triggering the respective doorbell, which includes updates to the sDMA pointer and memory barriers to ensure proper ordering of pointer updates after queue writes. Additionally, the number of synchronization commands scales with the number of engines, as each engine must synchronize with the host to signal task completion, further increasing control costs. Therefore, designing efficient DMA-based collectives across all transfer sizes requires a careful balance between parallelizing transfers and managing these overheads.

Broadcast and swap commands, which consolidate two transfer operations into a single command, reduce the number of DMA engines required by approximately half, thus lowering the number of queue doorbell and synchronization commands. Furthermore, sDMA engines in AMD Instinct™ MI300x can process multiple copy commands concurrently when scheduled back-to-back to the same queue as long as they are independent (no dependencies between their source

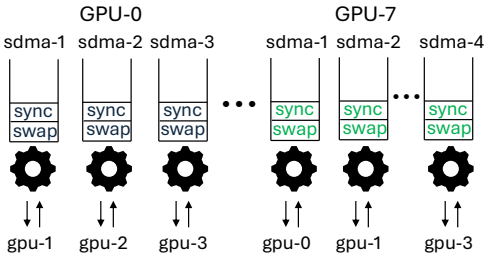

Figure 9. Swap-based alltoall (swap).

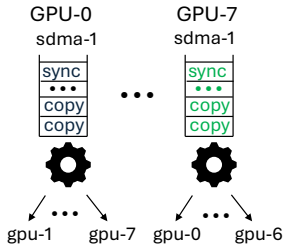

Figure 10. Back-to-back copy-based collective (b2b).

and destination buffers), which can effectively achieve parallelism similar to using multiple sDMA engines. Despite requiring multiple copy commands, scheduling them back-to-back reduces the number of engines needed, helping reduce the number of synchronization commands and thus the control phase time. In addition, it also require fewer scheduling (doorbell) events and can reduce the schedule phase time. Finding the optimal balance between the number of back-to-back commands in a single engine and the number of parallel engines is crucial for performance and may vary depending on the transfer sizes being exercised by the collective.

4.1.3 Off-the-Critical Path Launch. Another critical factor that influences collective performance is whether the commands can be scheduled in advance. By understanding the communication requirements and buffers ahead of time, it is possible to prepare the commands for execution and trigger them later, effectively hiding the associated control and scheduling phases. This approach allows only the execution of copy and synchronization commands to appear in the critical path. sDMAs provide mechanisms, such as the dependent or *poll* commands, to enable this optimization. A poll command takes a memory location and an expected value as input, and polls the memory location until it observes the expected value before retiring and allowing for subsequent commands in the queue to execute. This command can be used for fine-grained synchronization or to schedule copy operations (or other tasks) behind the poll. With this approach, the sDMA packet creation, queuing, and doorbell/fetching can occur off the critical path, ahead of time. The host can then trigger the execution by setting the poll's memory location to the expected value.

4.2 DMA Collective Implementations

Using the factors and strategies described in Section 4.1, we next design multiple optimized sDMA implementations for allgather (AG) and alltoall (AA) collectives.

4.2.1 Parallel Copy-Based Collective (`pcpy`/`baseline`). Since ML collectives such as AG and AA require multiple independent peer-to-peer transfers (i.e., $n * (n-1)$ where n is the number of devices) and given each GPU supports multiple sDMA engines which can be engaged in parallel (Section 2.4), a simple implementation of the collective entails each sDMA engine executing one independent copy as shown in Figure 7. To notify its completion of the copy and synchronize, each sDMA engine in addition also must execute a synchronization (sync) command which is an atomic add to a system memory location polled by the host. The collective is considered complete once the host observes completion across all sDMAs. Thus with n devices, this implementation uses $n * (n-1)$ engines, each with a copy and sync command. We term this implementation as 'parallel copy' or `pcpy` collective and use this as the baseline for comparison with our optimized sDMA collective implementation. Both AG and AA can use this implementation; in AG the source memory location across all copies within a device remains the same.

4.2.2 Broadcast-Based Allgather Collective (`bcst`). Since the AG collective requires each GPU to transfer its entire sub-array to every other GPU (Figure 2), it is well-positioned to take advantage of the *broadcast* commands and reduce the total number of sDMA commands and engines required as described in Sections 4.1.1 and 4.1.2. Since the broadcast commands support transfer data to a maximum of two destinations, collective implementations require each GPU to issue $(n-1)/2$ broadcast commands if n (number of devices involved) is odd or $(n-2)/2$ broadcast and an additional copy command if n is even. As shown in Figure 8, this ~halves both the number of commands as well as the number of sDMA engines required (and thus sync commands) as compared to `pcpy`. We term this implementation as the `bcst` collective.

4.2.3 Swap-Based Alltoall Collective (`swap`). Unlike AG, in AA, a GPU starts with a complete array and sends unique data to each of the other GPUs as shown in Figure 2. Thus, it cannot take advantage of the broadcast command. However, since the source buffers exchanged by each GPU pair is unique, it can leverage the *swap* command and reduce the total number of sDMA commands and engines required as described in Sections 4.1.1 and 4.1.2. AG cannot use swaps without requiring extra local memory-to-memory copies. AA with swaps exactly halves the number of commands as compared to `pcpy` as each of the two copies are replaced by a single swap command. Furthermore, as shown in Figure 9, to balance the number of commands and sDMA engines

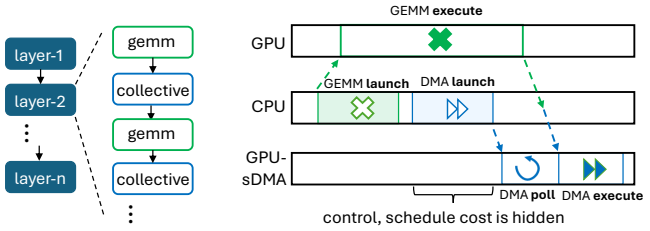

Figure 11. Leveraging deterministic communication patterns in ML to prelaunch collectives.

used between all GPUs, each GPU only executes ~half the number of swaps it requires, while the others are issued by the GPUs it requires the remaining swaps with. Note that since the CPU host issues these commands and they are processed in parallel by sDMA engines across all GPUs, there is no performance advantage of balancing, but can limit the power expended per GPU. We term this implementation as the `swap` collective.

4.2.4 Back-to-back Copies-Based Collective (`b2b`).

All the peer-to-peer transfers in AG and AA are completely independent and operate on distinct arrays and memory buffers. Thus, both stand to benefit from concurrent processing of back-to-back (b2b) *copies* on the same sDMA engine described in Section 4.1.2 which can help reduce the number of sDMA engines and therefore synchronizations commands.

While there are several implementations possible by sweeping the number of back-to-back copies per engine and the number of engines, we only consider implementations which further reduce the total commands and engines as compared to `bcst` and `swap` (it cannot be combined with `bcst` and `swap` implementations). Thus, we implement AG and AA collectives which use just one engine per device GPU, and schedule all copies required per GPU in a back-to-back manner as shown in Figure 10. As shown, this reduces the number of sync commands to only one per GPU. We term this implementation as the `b2b` collective. We also considered other intermediate implementations where multiple engines per device are used with fewer b2b copies, but they provide limited additional benefits.

4.2.5 Prelaunched Collectives (`prelaunch_`).

ML models (e.g., LLMs) have multiple uniform layers, each of which manifests as a sequence of GEMM and collective (and other smaller elementwise, activation) operations as shown in Figure 11. Furthermore, ML frameworks usually pre-allocate buffers and re-use them throughout the model's execution. Thus, this ahead-of-time information about the operator sequence and memory addresses can be leveraged to schedule collectives in advance and hide most of the control and schedule phase as described in Section 4.1.3.

There are multiple implementations possible when prelaunching; using a unique poll memory location for each DMA engine or for each GPU (all DMA engines within the

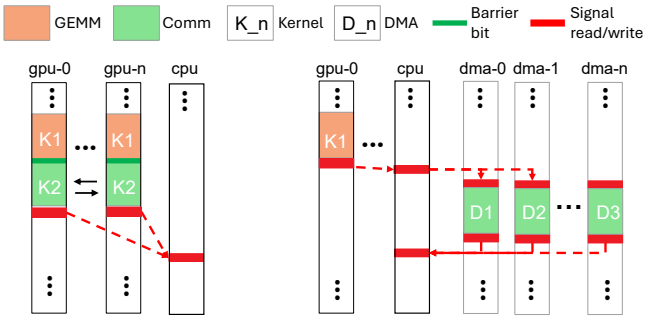

Figure 12. Producer GEMM to collective synchronization with (left) CU-based and (right) DMA-based communication

GPU poll on same memory) or one poll location for the entire collective. While more unique poll locations decrease serialization if any on the memory reads from sDMAs, they increase the cost of the CPU host's signaling time - a process atomically updates many memory location. For our evaluation we consider a unique poll location per DMA engine for all the four implementations discussed above to create `prelaunch_pcpy`, `prelaunch_bcst`, `prelaunch_swap` and `prelaunch_b2b`. Similar to the final synchronization, the memory location for poll also resides in the system memory and is updated by the CPU host to initiate the collective.

4.3 Additional Producer-Consumer Synchronization in Applications

AG and AA collectives in ML applications are used for communicating different type of tensors and have different types of dependencies with other operations based on the distributed technique deployed. In this section, we describe additional synchronizations that may be necessary as DMA collectives are integrated in end-to-end ML models.

4.3.1 Independent Collectives:

AG in FSDP gathers weights of layer(s) from multiple devices in both forward (inference and training) and backward pass (training only) of the model. During inference, these weights remain unchanged and during training these weights are updated in the previous iteration of the model, well ahead of when are used. Furthermore, their AG is scheduled (usually for a layer or set of layers at a time) ahead-of-time and on an independent stream as compared to the kernels which consume them. Therefore, the synchronization required to ensure the collectives complete before the consumer kernel starts is usually not on the critical path. In such cases, host-based trigger and synchronizations of DMA collectives discussed thus far is sufficient.

4.3.2 Dependent Collectives:

There are other scenarios where collectives directly operate on data produced by kernels (AA operates on activations generated by MoE's routing kernel) [22] or generate data that are consumed by kernels immediately (AG gathers all

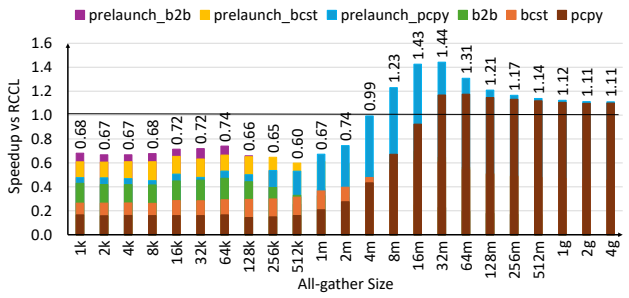

Figure 13. Speedup of sDMA Allgather collective optimizations vs. RCCL

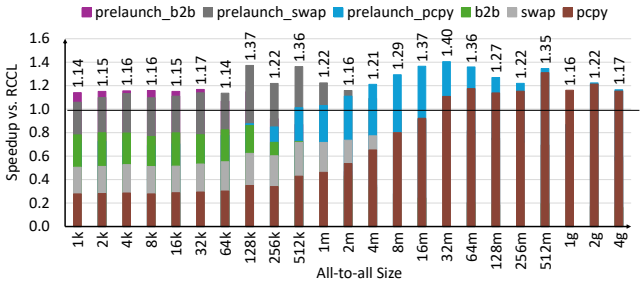

Figure 14. Speedup of sDMA Alltoall collective optimizations vs. RCCL.

activations for layers in Sequence Parallelism) [19]. In such scenarios, collectives require synchronization with these producer/consumer kernels (e.g., GEMMs) and such synchronizations are on the critical path.

As shown in Figure 12(left), CU-based collective kernels achieve this implicitly by being scheduled before or after the GEMM kernels on the same stream and by being processed by the same engine (i.e., command processor). Kernel packets have a barrier bit which when set (default by the GPU runtime) informs the scheduler to schedule a given kernel only after all prior kernels on the same stream complete [3]. DMA commands on the other hand are processed by different engines as compared to kernels and furthermore, some collective implementations require multiple engines per GPU (Sections 4.2.1- 4.2.3). Therefore, DMA collectives can require explicit synchronization with GEMMs in end-to-end applications which can add additional overheads. We consider existing mechanisms available for this kernel→DMA synchronization. As shown in Figure 12(right) this synchronization can be achieved with system memory-based signals and the CPU host. The GEMM kernel's stream executes a signal write command (possible with a hipStreamWrite32/64 [4]) after the GEMM completes. The CPU host waiting on this signal observes the write and issues another signal write which enables a DMA poll. While kernels can directly trigger DMA poll commands or poll on DMA sync commands, this requires changes to GEMM kernels which is complex as BLAS libraries contain several optimized implementations, each tuned for a given size [5]. Nevertheless, we leave the exploration of tighter synchronizations between kernels and DMA operations to future work.

5 Evaluation

5.1 System Setup

We evaluate the DMA collective implementations on an AMD MI300x Infinity Platform with eight AMD Instinct™ MI300x accelerators as described in Section 2.4. These implementations are also applicable for collectives with fewer GPUs (discussed in Section 6.5). We implement these collectives using the ROCt library which provides user-level APIs to create DMA packets and interact with DMA engines through the GPU driver. For CU-based collectives, we use AMD's state-of-the-art RCCL [9] library which has been tuned for each message size. We adjust the appropriate environment variables to scale the number of processes, enable performant algorithms (MSCLL [12], MSCCL++ [25]) and enable hipGraphs to ensure good performance. To evaluate synchronization with producer kernels, we use GEMM kernels from AMD's state-of-the-art BLAS library rocBLAS [10]. To evaluate power, we use an internal power-logging tool with a sampling interval of one millisecond. Samples are captured at different points during the collectives' execution using multiple runs and random delays, following the methodology described in prior work [27].

5.2 Collectives and Sizes Studied

We design and evaluate collectives for a range of sizes that collective libraries support. These include all-gather and all-to-all ranging from 1KB to 4GB. The small (KB) message sizes are important for ML inference scenarios, especially in the LLM decode phases where LLMs operate on / generate few tokens worth of activations at a time. The medium and large (MB-GB) collective sizes are common in both training and inference (prefill) and are required for communicating weights and activations depending on the distributed setup.

5.3 Collective Performance

To evaluate our optimized DMA collectives, we compare their performance with the `baseline` or `pcpy` DMA implementations from prior works. But since the goal of DMA collectives is to provide an alternative to CU-based collectives for concurrent compute and communication scenarios (and free up CU and cache resources), we mainly focus on how they perform as compared to their best CU-counterpart from RCCL. Thus, in Figures 13 and 14 we plot the ratio of CU to DMA collective execution times, or the speedup of DMA collectives over CU-based ones. For both allgather (AG) and alltoall (AA), we show `pcpy`, `b2b`, and their `prelaunch_`

Table 1. Performant implementation for allgather collective.

Size range	Allgather Optimizations
1KB <= size <256KB	b2b, prelaunch
256KB <= size <1MB	bcst, prelaunch
1MB <= size <512MB	pcpy, prelaunch
>= 512MB	pcpy

Table 2. Performant implementation for alltoall collective.

Size range	Alltoall Optimizations
1KB <= size <64KB	b2b, prelaunch
64KB <= size <4MB	swap, prelaunch
4MB <= size <1GB	pcpy, prelaunch
>= 1GB	pcpy

variants. In addition, for AG we show `bcst` and AA we show `swap` along with their `prelaunch_` variants.

We show speedups of different variants stacked to depict how each of the DMA optimizations help inch closer to CU-based collective performance. For example, in Figure 13, the stacked speed ups of `pcpy` and `b2b` show that `b2b` achieves higher speedups than `pcpy`. However, note that, `pcpy` is not required for `b2b` to achieve this speedup. We discuss each of these DMA implementations in detail below:

5.3.1 `pcpy` **or** `baseline`**.** The AG and AA `pcpy` implementation outperforms CU-based collectives by 14% and 18% geomean for sizes greater than 32MB. This is a result of lower metadata with DMA transfers which improves network BW efficiency. However, `pcpy` variants are 4.5× and 2.5× slower in the remaining, smaller sizes. This is because of the additional non-copy phase times (control, schedule, and sync in Section 3) from engaging several (56) DMA engines that dominate execution at these sizes. While the sync commands (e.g., atomics) execute in parallel, control and schedule are serialized within a single CPU process. We discuss multi-process executions in Section 6.3.

5.3.2 `bcst`**.** As shown in Figure 13, `bcst` speeds up AG collective by 1.7× geomean over `pcpy` for sizes up to 4MB, narrowing the gap vs. CU-based RCCL from 4.5× to 3× for sizes up to 32MB. This results from using ~half the number of commands and engines, as broadcast commands can transfer to two different destinations (Section 4.2.2). As sizes increase and the non-copy phases contribute less to end-to-end time, `bcst`'s benefits decrease. At larger, bandwidth-bound sizes, `bcst` does not provide additional benefits as the parallel engines within each GPU in `pcpy` are able to saturate the total network bandwidth of a single GPU. Overall, while `pcpy` is sufficient for >32MB AG collective, `bcst` is required for up to 4MB for improved performance.

5.3.3 `swap`**.** Similar to `bcst`, `swap` in Figure 14 speeds up AA collective by 1.7× geomean over `pcpy` for sizes up to 4MB, and reduces the performance gap vs. RCCL from 2.5× to 1.6× for sizes up to 32MB. Similar to `bcst` this results from using half the number of commands and engines as each swap command performs two copies worth transfers (Section 4.2.2). Overall, while `pcpy` is sufficient for >32MB AA, `swap` is required for up to 4MB for improved performance.

5.3.4 `b2b`**.** While `bcst` and `swap` help, there is still a considerable 3× and 1.6× geomean performance gap between DMA and RCCL collectives for AG and AA, respectively. Our DMA implementation, `b2b`, with all copies from a GPU scheduled back-to-back on a single DMA engine (Section 4.2.4) further bridges this gap by speeding up AG collectives by 2.7× geomean over `pcpy` and by 1.5× geomean over `bcst` for <1MB. Similarly, for AA collectives `b2b` is 2.5× faster than `pcpy` and 1.4× faster than `swap` for <1MB. Using a single DMA engine reduces non-copy times (fewer sync commands, fewer doorbells) by ~7× and ~4× compared to `pcpy` and `bcst`/`swap`, respectively, with negligible impact on copy times.

`Bcst` and `swap` however remain the superior choice for 1-4MB AG and AA, providing 1.4× and 1.7× geomean speedups over `b2b`. Similarly, `pcpy` remains performant at sizes >4MB. This is because the increase in total copy time (from interleaving and overlapping seven back-to-back copies) exceeds the benefits from reduced non-copy times in `b2b`. This also demonstrates that across the size spectrum evaluated, each of these implementations benefit unique size ranges and thus are important. Overall, as shown in Figures 13 and 14, including `b2b` reduces performance gap vs. CU-based RCCL from 3× and 1.6× to 2.3× and 1.3× geomean for sizes up to 32MB for AG and AA.

5.3.5 `prelaunch_`**.** Finally, we evaluate the impact of prelaunching and hiding the non-copy costs (Section 4.2.5). We show its impact on each of the variants; as an example `prelaunch_b2b` has prelaunch applied on top of the `b2b` implementation such that each DMA engine on the GPU has a poll command followed by copy and sync. As shown in Figures 13 and 14, pre-launching benefits the entire size range. Intuitively, it has the highest impact on implementations which use more commands and DMA engines and have longer non-copy phases; it speeds up `pcpy` by 1.9×, both `bcst` and `swap` by 1.5×, and `b2b` by 1.2× geomean across the size range. Despite the higher benefit with `pcpy`, prelaunched `b2b`, `bcst` and `swap` still outperform `prelaunch_pcpy` at sizes <1M for AG and < 4M for AA. `prelaunch_pcpy`, however, is required for improved performance at larger sizes; it speedups up 1MB-256MB AG by 1.5× geomean and 4MB-512MB AA by 1.3× over the best of `pcpy`, `bcst` and `swap`.

Overall, starting with a 4.5× and 2.5× geomean slowdown compared to RCCL for <32MB AG and AA, respectively, our optimized sDMA collectives bring down the slowdown to 30% geomean for AG and are 20% faster for AA. They also achieve 20% speedup over RCCL in the larger, 32MB-1GB,

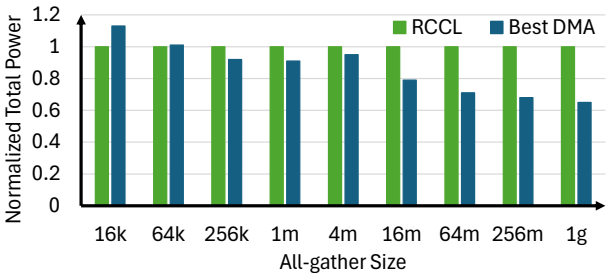

Figure 15. Total power consumed by best DMA vs CU (RCCL) collectives

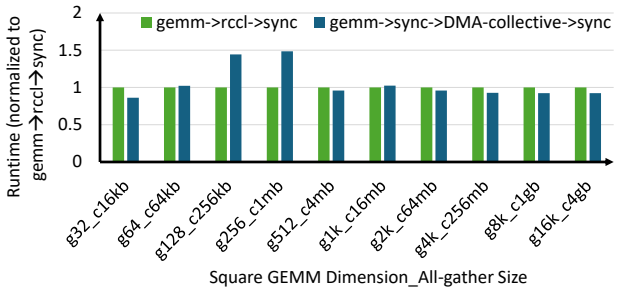

Figure 16. Producer GEMM→collective time with CU (RCCL) vs DMA collective. gX_cY represents a square GEMM with all dimension as X followed by a collective of size Y.

size range. We list the best-performing implementations for different size ranges in Tables 1 and 2.

5.4 Collective Power

Offloading communication to DMAs and freeing up GPU compute resources also stands to provide power savings. Figure 15 shows the total GPU power (including XCD, IOD and HBM as detailed in Section 2.4) consumed by AG collective and compares best-performing DMA-based implementation with CU-based ones from RCCL. It shows that DMA collectives consumes ~32% less power than its CU counterpart for larger, >=64MB sizes. This benefit is largely a result of no CU activity and thus considerably (3.7×) less XCD power with DMA collectives. DMA collectives also have smaller AID and HBM power as they use peer-accessible buffers and avoid additional local memory accesses to intermediate buffers as in RCCL (more in Section 6.4). Since DMA AG is ~20% faster than RCCL at these sizes, this translates to energy savings as well. The power benefit decreases at smaller sizes as RCCL stresses both CUs and memory resources less at these sizes. Nevertheless our optimized `prelaunch_b2b` collectives helps limit power by using fewer engines (3-4% lesser power) than `prelaunch_pcpy` at the 16-64KB range. Finally, `prelaunch_bcst` also provides power savings (5-10%) due to reduced memory traffic (broadcast reads once and writes to two destinations) at >1MB sizes.

5.5 Producer-Consumer Synchronizations

We also evaluate the cost of additional synchronizations DMA collectives require as discussed in Section 4.3. Figure 16 compares the execution times of producer→collective, where the producer is a GEMM kernel generating data consumed by the collective, which is either a RCCL or DMA-based allgather (AG) operation. They require different types of synchronizations as shown in Figure 12.

As shown, for AG size >4MB, the process with DMA still achieves similar or better performance as compared to the one with CU-based RCCL. For smaller sizes, the performance reflects the differences in the standalone collective performance shown in Figure 13. Thus, in-spite of requiring additional synchronization and coordination between multiple queues, performant DMA collective stand to achieve similar performance in end-to-end applications while making CUs available for other independent operations. Note that the execution times presented capture time from start of the GEMM kernel execution to when the CPU observes the final signal in Figure 12. While the overhead of issuing multiple DMA and signal write APIs is larger than RCCL kernels', this overhead typically occurs in the shadow of other operation executions in end-to-end applications or can be reduced through the use of graph launches (e.g., HIP graphs [2]).

6 Opportunities and Future Work

6.1 Additional Collectives (Reduce-scatter)

Reduce-scatter (RS) collective requires both communication and compute. Since DMAs lack compute support, prior works have explored communication using DMAs and reduction using CUs for RS [25]. While this still uses CUs, they are required for a much shorter duration (only the final reduction). Furthermore, since these implementations are similar to `pcpy`, their performance suffers at KB-MB range. Optimizations presented in this paper can accelerate the communication phase of RS in such hybrid mechanisms and can help outperform CU-only collectives when overlapped.

6.2 Additional Memory Benefits

The optimized collectives `bcst` and `swap` have additional memory benefits which can reduce memory interference as compared to `pcpy` and CU-based collectives. Broadcast commands read the source data once and send it to two destination GPUs. This reduces redundant memory accesses. `Swap`, on the other hand has similar reads and writes as compared to `pcpy` and CU-based implementations. However, `swap` enables in-place data exchanges whereas `pcpy` and CU-based do not. The latter add memory capacity pressure by either requiring out-of-place collectives with separate input and output buffers, or by requiring intermediate buffers to prevent overwriting of data during the exchange. The latter would also involve additional local memory copies and thus

higher memory interference with compute operations. Using DMA `swap` commands helps avoid these.

6.3 Number of CPU Processes

While our DMA implementations use a single CPU process to initiate collectives, employing multiple processes can help parallelize and further reduce the non-copy time with multiple copies (Section 5.3.1). However, these processes can become a bottleneck when nodes are shared among multiple applications. The is compounded when using a single process for each copy or engine; `pcpy` would require $n \times (n-1)$ processes (56 processes for an 8-GPU node). Nevertheless, a balanced implementation that uses a single process per GPU can further accelerate our DMA collectives.

6.4 Registered Memory Allocations

The DMA implementations discussed in this work use buffers that are directly accessible by peers. In end-to-end applications, this necessitates that user or application buffers possess similar capabilities. Collective libraries offer buffer registration APIs to enable this [6] but when unavailable, collectives rely on intermediate buffers and additional memory copies to stage data in those buffers. Our DMA collectives can be extended to support such intermediate buffers if needed. Since our implementations utilize at most seven DMAs per GPU, these memory copies can be pipelined and overlapped with remote copies using the remaining spare DMA engines (see Section 2.4). However, similar to CU-based implementations, this approach requires additional synchronization between the local and remote copies.

6.5 Other Topologies and Scaling GPU count

While the DMA implementations in this work use a one-shot (direct) approach and leverage the fully-connected topology, ring-based implementations are important in other topologies. Ring algorithms also have the advantage of using smaller intermediate buffers (if required as described in Section 6.4) as they only communicate one chunk or sub-array at a time, unlike in one-shot. While DMA collectives do not require such intermediate buffers, they can nevertheless also use ring implementations and provide similar performance at larger sizes. Finally, while we use an eight-GPU node for our evaluations, the optimizations are valid for other node sizes as well. In fact they stand to provide larger benefits as GPU count increases as non-copy phases scales with peers/copies.

6.6 Collective Libraries with DMA

Modern collective libraries incorporate various algorithms, synchronization protocols, and additional features optimized for different types of collectives, sizes, and topologies. We envision that DMA usage should also be considered a key tuning parameter for these libraries. The optimized DMA implementations proposed in this work can be invoked when overlap is necessary and/or can be selected to further enhance both performance and power efficiency, as demonstrated in this work.

7 Related work

7.1 Communication Offload

Works which improve overlapped compute and communication performance usually offload communication to dedicated accelerators on GPUs, requiring extensive hardware support. Some use custom accelerators which frees up CUs for concurrent compute and reduces memory interference by buffering intermediate data [24]. Similarly, compute-capable switches help offload reduction operations in collectives (such as in allreduce) which helps limit traffic over network links and memory sub-system [18]. Switch offloads, however, still require CUs to orchestrate data movement and in-switch commands. We consider offload but by leveraging *existing* DMAs, requiring no hardware changes.

7.2 DMA Offload

Several works offload communication to DMAs. MSCCL++ [25] initiates DMAs from GPU kernels (through proxy channel in CPU) involving CUs, which in our work we prefer to reserve only for compute operations. Furthermore, they implement only the `pcpy` variant for collectives which along with additional indirection to initiate DMA copies can hurt KB-MB sized collective performance. ConCCL [7] shows DMA's potential to accelerate concurrent compute, however, focuses on large collectives. ARK [16] focuses on small-sized DMA transfers but proposes a GPU thread-initiated DMA prototype which requires considerable software and hardware changes. This work in contrast relies on existing software stack and HW to build performant DMA collectives and, unlike other works, evaluates both performance and power.

7.3 Fine-grained DMA Offload

Several works leverage DMAs for fine-grained compute-communication overlap [14, 15, 33]. Some of these only require a GPU to perform a single peer-to-peer copy to another GPU rather than a collective which is not performant on an MI300x node as they under utilize its network bandwidth [14, 15]. Others rely on `pcpy`-like variant which under performs for KB-MB sizes [33]. Other works require hardware modifications to initiate DMA transfers [21]. Nevertheless, insights from this work can benefit such fine-grained mechanisms involving small-size transfers/collectives.

8 Conclusion

Mechanisms to effectively overlap and hide communication are critical for distributed ML performance. This work focuses on efficiently offloading small-sized ML collectives to *existing* DMA engines to reduce their resource interference with compute when overlapped. We first provide an in-depth

latency breakdown of a DMA offload which guides our design of DMA collectives. Next, we leverage existing DMA capabilities of an AMD Instinct™ MI300X GPU to accelerate DMA-based ML collectives. Starting with a baseline implementation which is 4.5× and 2.5× slower than RCCL for <32MB sizes, we bridge this gap to only 30% slower and 20% faster for allgather and alltoall, respectively. For larger sizes, DMA collectives are ∼20% faster and also consume ∼32% less power compared to compute-unit (CU)-based RCCL. These performant DMA collectives further stand to enable efficient computation and communication overlap and improve end-to-end ML performance.

Acknowledgment

AMD, the AMD Arrow logo, AMD Instinct, AMD ROCm, AMD Infinity Cache, AMD Infinity Fabric, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] 2014. DMA Packets. https://people.freedesktop.org/~agd5f/dma_packets.txt.

[2] 2024. HIP Documentation. https://rocm.docs.amd.com/_/downloads/HIP/en/docs-6.1.2/pdf/.

[3] 2024. ROCR Documentation. https://rocm.docs.amd.com/_/downloads/ROCR-Runtime/en/master/pdf/.

[4] 2025. AMD ROCm documentation. https://rocm.docs.amd.com/en/latest/.

[5] 2025. Tensile Documentation. https://rocm.docs.amd.com/_/downloads/Tensile/en/latest/pdf/.

[6] 2025. User Buffer Registration. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/bufferreg.html.

[7] Anirudha Agrawal, Shaizeen Aga, Suchita Pati, and Mahzabeen Islam. 2025. Optimizing ML Concurrent Computation and Communication with GPU DMA Engines. arXiv:2412.14335 [cs.AR] https://arxiv.org/abs/2412.14335

[8] AMD. [n. d.]. *ROCm Runtime (ROCr)* . https://github.com/ROCm/ROCR-Runtime

[9] AMD. 2025. *ROCm Communication Collectives Library (RCCL).* https://github.com/ROCm/rccl

[10] AMD. 2025. ROCm/rocBLAS: Next generation BLAS implementation for ROCm platform. https://github.com/ROCm/rocBLAS.

[11] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. 2024. Flux: Fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858* (2024).

[12] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. Msccclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 502–514.

[13] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[14] Ali Hassani, Michael Isaev, Nic McDonald, Jie Ren, Vijay Thakkar, Haicheng Wu, and Humphrey Shi. 2024. Distributed GEMM.

[15] Horace He, Less Wright, Luca Wehrstedt, Tianyu Liu, Wanchao Liang. 2024. Introducing Async Tensor Parallelism in PyTorch. "https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487".

[16] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. 2023. {ARK}:{GPU-driven} Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 87–101.

[17] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 402–416. doi:10.1145/3503222.3507778

[18] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, IEEE Computer Society, Washington, DC, USA, 996–1009.

[19] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[20] NVIDIA. [n. d.]. *NCCL*. https://github.com/NVIDIA/nccl

[21] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. 2024. T3: Transparent Tracking & Triggering for Fine-grained Overlap of Compute & Collectives. In *Proceedings*

of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 1146–1164.

[22] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*. PMLR, 18332–18346.

[23] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[24] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. 2021. Enabling Compute-Communication Overlap in Distributed Deep Learning Training Platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, IEEE Press, Piscataway, NJ, USA, 540–553. doi:10.1109/ISCA52012.2021.00049

[25] Aashaka Shah, Abhinav Jangda, Binyang Li, Caio Rocha, Changho Hwang, Jithin Jose, Madan Musuvathi, Olli Saarikivi, Peng Cheng, Qinghua Zhou, Roshan Dathathri, Saeed Maleki, and Ziyue Yang. 2025. MSCCL++: Rethinking GPU Communication Abstractions for Cutting-edge AI Applications. *arXiv preprint arXiv:2504.09014* (2025).

[26] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019), 9 pages. arXiv:1909.08053 [cs.CL] http://arxiv.org/abs/1909.08053

[27] Varsha Singhania, Shaizeen Aga, and Mohamed Assem Ibrahim. 2025. FinGraV: Methodology for Fine-Grain GPU Power Visibility and Insights. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 96–107.

[28] Alan Smith, Eric Chapman, Chintan Patel, Raja Swaminathan, John Wuu, Tyrone Huang, Wonjun Jung, Alexander Kaganov, Hugh McIntyre, and Ramon Mangaser. 2024. 11.1 AMD InstinctTM MI300 Series Modular Chiplet Package – HPC and AI Accelerator for Exa-Class Systems. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 67. 490–492. doi:10.1109/ISSCC49657.2024.10454441

[29] Alan Smith, Gabriel H. Loh, John Wuu, Samuel Naffziger, Tyrone Huang, Hugh McIntyre, Ramon Mangaser, Wonjun Jung, and Raja Swaminathan. 2024. AMD Instinct™MI300X Accelerator: Packaging and Architecture Co-Optimization. In *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 1–8. doi:10.1109/VLSITECHNOLOGYANDCIR46783.2024.10631545

[30] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. 2022. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 93–106. doi:10.1145/3567955.3567959

[31] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. 2025. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts. arXiv:2502.19811 [cs.DC] https://arxiv.org/abs/2502.19811

[32] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arXiv:2304.11277 [cs.DC] https://arxiv.org/abs/2304.11277

[33] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. 2025. Triton-distributed: Programming Overlapping Kernels on Distributed AI Systems with the Triton Compiler. arXiv:2504.19442 [cs.DC] https://arxiv.org/abs/2504.19442