Optimizing Distributed ML Communication with Fused Computation-Collective Operations

Kishore Punniyamurthy, Khaled Hamidouche, Bradford M. Beckmann
AMD Research and Advanced Development
Kishore.Punniyamurthy@amd.com, khaledhamidouche@gmail.com[§], Brad.Beckmann@amd.com

***Abstract*—Machine learning models are distributed across multiple nodes using numerous parallelism strategies. The resulting collective communication is often on the critical path due to a lack of independent coarse-grain computation kernels available to execute.**

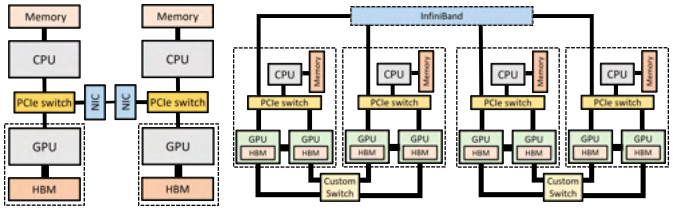
In this work, we propose fusing computation with its subsequent collective communication and leverage GPUs' massive parallelism, along with GPU-initiated communication, to overlap communication and computation. Specifically threadblocks/workgroups (WGs) immediately communicate their results to remote GPUs after completing their computation,while other WGs within the same kernel perform computation. We developed three prototype fused operators (embedding+All-to-All, GEMV+AllReduce, and GEMM+All-to-All) to address the communication overheads in DLRM, Transformers and MoE model architectures. We expose fused kernels as new PyTorch operators, as well as extend the Triton framework to demonstrate their practicality. Our evaluations show our approach effectively overlaps communication with computations, subsequently reducing their combined execution time achieving 12% - 31% lower execution time across all three operators.

***Index Terms*—Collective communication, distributed ML, DLRM, GPU, MoE, Transformers**

I. Introduction

Machine learning is currently being used across a wide variety of applications ranging from classification (e.g., fraud detection, medical diagnostics, facial recognition) and pattern analysis (e.g., product recommendations, stock price prediction) to content generation (e.g., code generation, chatbots, image/video generation). Machine learning (ML) models are increasing in size to tackle more complex problems. Studies [58] have shown that the model sizes have increased by five orders of magnitude between 2018 to 2022. Large ML models have consequently fueled the development of distributed systems capable of meeting their capacity and compute requirements. Subsequently, parallelization techniques have been developed to efficiently map ML models on to distributed systems [11]. The resulting communication in distributed ML models (e.g., weight updates, activation exchanges between layers, etc.) are difficult to hide due to absence of independent computation and thereby become a significant bottleneck [40], [48] as the system scale grows.

ML trends are also influencing system design and architecture development. Figure 1 compares a traditional CPU-GPU multi-node system to a state-of-the-art node design optimized for HPC/ML. Other than compute and memory improvements,



(a) Traditional node design (b) Emerging HPC/ML node design

Fig. 1: System architecture trends.

we notice two broad trends. Firstly, intra- and inter-node communication bandwidth and latency have been improved using proprietary interconnects [4], [17] and high-bandwidth network interface cards (NICs). Secondly, given the increased preference to use GPUs for ML workloads, GPUs are becoming the primary computation engine tightly integrated with on-node CPUs, GPUs and are directly communicating with high-radix switches [15] or direct-attached NICs [2], [26]. However, applications have yet to fully embrace these trends and often rely on the CPU [35], [47] to perform remote communication. While progress has been made to remove CPUs from the data path, for example, GPUDirect RDMA [6], [9] enables direct data transfers between GPU memory and a NIC, communication is still triggered by the host CPU and often at kernel boundaries. Such approaches are mostly suitable for bulk-synchronous applications where communication latency can be hidden with independent computation at the kernel granularity and the kernel-launch overhead is amortized using large kernels.

In this paper, we aim to address the growing bottleneck of exposed collective communication by fusing and overlapping communication and dependent computation using intra-kernel GPU-initiated networking. GPU ML compute kernels are often implemented such that each threadblock/workgroup[1] (WG) computes an independent portion of the output, but current ML frameworks separate computation and communication into separate kernels. Our work leverages the independent nature of WG execution to overlap computation with collectives by communicating outputs of individual WG (or WG clusters) asynchronously using GPU-initiated intra-kernel communication. Recent GPU micro-architectural features (e.g., GPU cache flush instructions [5], threadblock cluster synchronization [16]) have made GPU-initiated networking operations performant

[§]Author is no longer part of AMD Research and Advanced Development

[1]Workgroup is the OpenCL equivalent of CUDA threadblock

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

and thus an attractive option for collective communication.

Our approach is easy to apply and leverages the existing workgroup-level work partitioning in GPU applications and performs well on current hardware. We have developed self-contained, persistent GPU kernels [46] where one or more logical WGs, upon completing their computations, issue non-blocking transactions to communicate their results to remote GPUs. By immediately scheduling non-blocking network transactions, data fragments are communicated as they are computed without waiting for kernel completion.

We demonstrate our approach by creating fused communication-computation kernels to address the collective bottlenecks in popular ML architectures, such as Deep Learning Recommendation Models (DLRM), Transformers, and Mixture of Experts (MoE). Specifically, we create representative prototype kernels which fuse embedding pooling, matrix-vector multiplication (GEMV), and matrix-matrix multiplications (GEMM) with collectives (All-to-All and AllReduce). Our approach can fuse both inter-node (e.g., RDMA) and intra-node GPU communication. For the locally communicated data, we develop zero-copy fused kernels where the GPU threads directly store their computed results to collective destination buffers. We also demonstrate the different ways fused operators can be integrated within ML frameworks. Specifically we expose the new fused communication-computation kernel as a new operator within PyTorch [20] and extend *Triton*, a Python-like language for developing highly efficient GPU code [28] in PyTorch [1], with the necessary communication primitives to develop custom fused kernels.

This paper makes the following key contributions:

- We propose a novel approach which breaks the kernel-boundary communication model by fusing computation and collective communication within the same kernel while adhering to the application's data dependencies. Further, we propose zero-copy fused kernels where the results are directly written to the peer GPU memory thereby eliminating intermediate buffering and copy operations.
- We develop three fused communication-computation prototype kernels to address the collective overhead in DLRM, Transformers, and MoE architectures. Specifically, we develop *embedding pooling + All-to-All*, *GEMV + AllReduce*, and *GEMM + All-to-All* fused operators. Our evaluations show that our fused operators achieve 31%, 13%, and 12% lower latency respectively.
- We implement two different approaches to integrate fused operators within ML frameworks. 1) We expose new fused communication-computation kernels as new operators within PyTorch that can be directly used by developers. 2) We extend the Triton framework to include new communication primitives to enable the creation of customized fused kernels for specific use cases. For example, we use our Triton extensions to develop customized *GEMM + All-to-All* kernels for the unique sizes seen in MoE layers.
- We evaluate the trade-offs for using GPU-initiated communication and highlight the factors critical (e.g., occupancy) to achieve effective overlap with low overhead.

II. Background

In this section, we first illustrate the communication bottleneck observed in several distributed ML applications: DLRM, Transformers, and MoEs. We then provide a background on GPU-initiated communication and intra-kernel networking.

A. Collectives in Distributed ML Models

Large ML models use parallelism strategies (e.g., tensor parallelism, model parallelism, fully-sharded data parallelism (FSDP) [8]) to avoid data duplication across distributed nodes. However, such strategies result in additional collective communication to train and execute the models.

DLRM (Embedding + All-to-All): Recommendation models are widely deployed for ranking and click through rate predictions tasks. Due to the large memory requirements of embedding tables, DLRMs typically employ model parallelism for distributing embedding tables (table, row, and column parallelism [48]) across GPUs [49]. The top multi-layer perceptron (MLP) layers of DLRM use data parallelism for scaling across multiple GPUs. In order to switch from model parallelism execution (for embedding operations) to data parallelism (for top MLP layers), All-to-All collective operations are used [49] as shown in Figure 2. Note that the bottom MLP layers are the only independent computation available to be overlapped with the All-to-All collective but they are usually small and thus cannot hide communication effectively. Prior research has shown that All-to-All latency is significantly exposed (>35%) and has a direct impact on the overall latency in state-of-the-art systems [48].

Hiding the All-to-All collective will require overlapping it with dependent computations (embedding pooling or top MLP), which is not possible with existing bulk-synchronous kernel execution. In this work, we overlap All-to-All communication with the embedding pooling operator by communicating fragments of the pooling output data as they are computed.

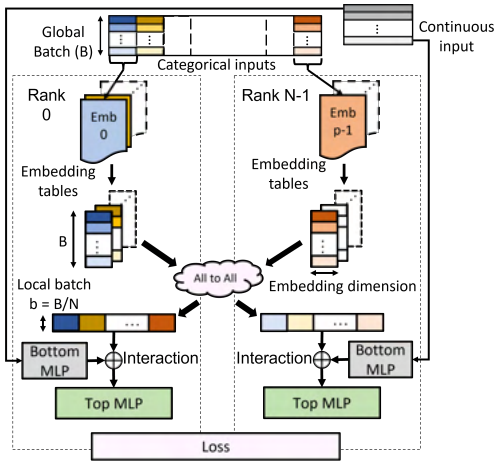


Fig. 2: DLRM forward pass.

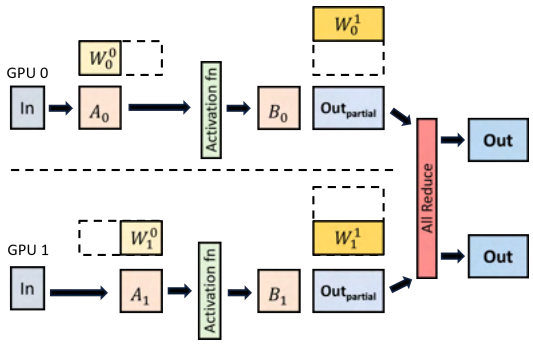


Fig. 3: Model parallelism in Transformer MLP layer [56].

Since the embedding pooling operation is parallelized across independently executing WGs, the output computed by a single WG or cluster of WGs can be communicated independent of other output without violating data-dependencies.

Transformers (GEMV + AllReduce): Transformers have become a popular ML model architecture for natural language processing, machine translation, and text generation tasks. Auto-regressive Transformers (e.g., GPT, Llama2) leverage the prior tokens and iteratively predict the next tokens. Their inference consists of two phases [50]: 1) *Prompt (or pre-fill) phase*: all input tokens are processed to generate the first token. 2) *Token (or decode) phase*: iterative phase where tokens generated previously along with the inputs are used to predict the next token. Typical models consist of a self-attention layer followed by feed-forward layer and this building block is repeatedly instantiated to create the entire model. The computations associated with these layers are matrix-matrix multiplications (during training, prompt phase of inference) or matrix-vector multiplications (token phase of inference) [51], and are usually combined with element-wise operations (e.g., normalization, addition).

Transformer architectures used in modern language models are too large to fit within a single GPU device and are thus partitioned. Figure 3 shows an example of model parallelism applied to a Transformer's feed forward layers as proposed in Megatron-LM [56]. The feed forward layer consists of two linear layers with an activation layer in-between. The weights corresponding to each of the MLP layers are partitioned across two GPUs, such that weights of the first layer are partitioned column-wise (W_0^0, W_1^0), while the weights of second layer are partitioned row-wise (W_0^1, W_1^1). The input is duplicated across both GPUs and multiplied with half of the weights (W^0). The outputs (A_0, A_1) are passed through the activation function before being multiplied with weights of second layer (W^1) to generate the partial outputs ($Out_{partial}$). Finally, the partial outputs are reduced using AllReduce to obtain the final results. This AllReduce operation contributes significantly (up to 46% [51]) towards the inference latency.

Due to the lack of independent computation, the AllReduce collective can only be overlapped with the preceding computation. Since inference latency is very dominated by the token-phase [50] which performs matrix-vector multiplication (assuming no batching), we focus on fusing the AllReduce with the GEMV computation. GPU kernels for the GEMV operation [25] typically divide the computation across the WGs, where each WG is responsible for computing a tile of the output vector. These output tiles can then be communicated and reduced independently of other tiles. We exploit this opportunity to develop a fused *GEMV + AllReduce* operator.

Mixture of Experts (GEMM + All-to-All): Mixture of Expert (MoE) [55] applications are growing in popularity as they allow for increasing the model parameters without linearly scaling the computation cost. MoE architectures replace dense feed-forward networks (FFN) using multiple sub-layers called *experts*. Figure 4 shows an example of an MoE layer with four experts. Each expert is an FFN by itself with two linear layers which translate into GEMM kernels. Effectively, the model now contains two weight matrices per expert (W_0^i, W_1^i where $i \in \{0..3\}$ in the example shown) which get trained on different subset of inputs. Inputs to MoE layers are passed to only a subset of experts as determined by a gating function (G in Figure 4). For example, assume that each input is only routed to two experts (top-2 routing) and that the gating function G results in a uniform distribution. An input batch size of 16 will result in each expert processing eight ($= 4 \times 2$) inputs.

MoE model architectures typically employ expert-level parallelism [52] [34] where the expert sub-layers are mapped across different GPUs. The example in Figure 4 shows the experts (weight matrices W_0^i, W_1^i where $i \in \{0..3\}$) distributed across four GPUs resulting in two All-to-All collectives, one for distributing the inputs across experts (All-to-All Dispatch) and another to gather the outputs from different experts (All-to-All Combine). Inputs from four GPUs (In_{0-3}) are distributed to the individual experts (assuming uniform distribution) based on the gating function (G) using All-to-All (dispatch). The All-to-All output is passed as input to individual expert feed forward layers (one per GPU) and the output is returned to the original GPU using All-to-All (combine). These All-to-All collectives are on the critical path and add significant overhead (up to 43% [40]) to the MoE execution time.

The All-to-All collectives can only be overlapped with dependent GEMM operations. Similar to GEMV, WGs of the

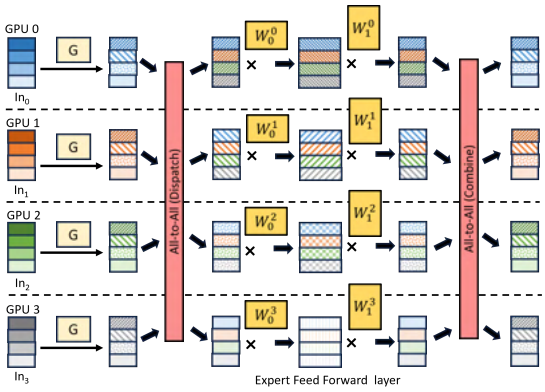


Fig. 4: MoE layers distributed across GPUs [40].

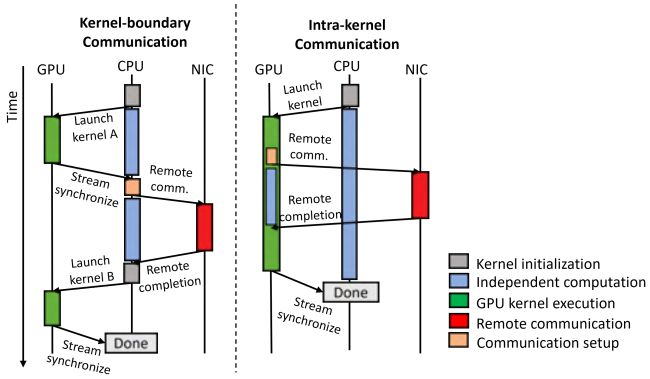


Fig. 5: Kernel boundary vs intra-kernel communication.

MoE GEMM kernels compute output tiles independently and thus can be communicated as soon as they are computed. In this work, we consider top-2 routing (i.e., each token is communicated to two experts), and we assume equal workload distribution across the experts. We demonstrate our approach by fusing MoE GEMM computation with All-to-All Combine communication.

B. GPU-initiated intra-kernel communication

GPU-based HPC and ML systems typically use GPU-Direct Remote Direct Memory Access (RDMA) [9] and associated programming models (e.g., CUDA-Aware MPI [6]) to enable direct data transfers between the GPU and the NIC. Bypassing CPU memory helps achieve better latency and thus these approaches are widely used in HPC [37], [44], and ML (RCCL [22], NCCL [14]). However, such communication is performed at kernel boundaries and typically by the host CPU.

GPU-initiated intra-kernel communication allows GPU threads to directly initiate communication with NICs and peer GPUs. Vendor-specific GPU libraries (e.g., NVSHMEM [18], ROC_SHMEM [24], MSCCL++ [13]) have been developed that enable applications to perform intra-kernel GPU-initiated communication. Moreover, recent GPU micro-architecture features are facilitating GPU-initiated networking further. For example, prior versions of ROC_SHMEM required data to be allocated as un-cacheable in order to prevent stale data from being communicated to remote nodes. However, the recent introduction of intra-kernel cache flush instructions [5] allows GPU threads to flush data to memory while the kernel executes. These new instructions enable the same kernel that previously computed and cached data to later initiate network transactions for that same data. Further, new programming abstractions, such as threadblock cluster [16], can enable faster coordination between WGs to perform intra-kernel networking at sizes efficient for communication.

Figure 5 compares and contrasts the system interactions during conventional kernel boundary communication versus intra-kernel communication. Kernel boundary communication requires the entire computation kernel to complete before the remote/network communication can be triggered (usually by the host CPU). The CPU then launches the dependent communication kernel. To overlap communication in the conventional approach, techniques such as double-buffering and GPU streams have been deployed. In the absence of independent work, applications are broken into smaller independent kernels where each smaller kernel is executed as a separate stream and computation of one stream is overlapped with the communication of another. This can result in large number of small kernel launches and add significant stream management overhead [45], [48], [61]. In contrast, GPU-initiated intra-kernel communication allows GPU threads to issue remote communication while other threads perform their computation, enabling fine-grained communication and computation overlap. Figure 5 illustrates a GPU directly communicating with a NIC [38], but an alternative form of intra-kernel communication would be GPU threads triggering communication using a CPU proxy thread [13].

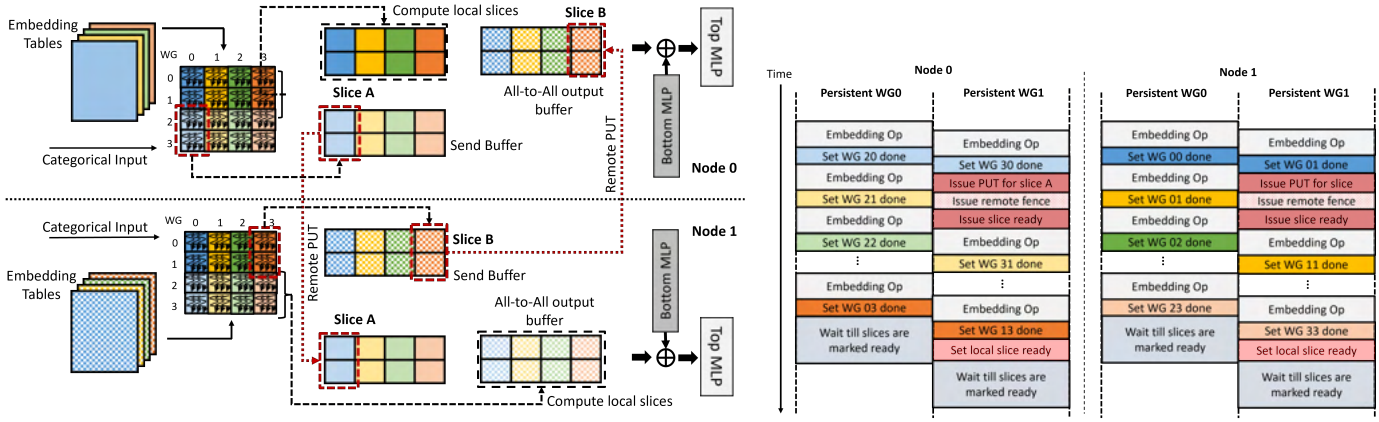
III. Fused Computation and Communication

In this section, we illustrate our fused communication + computation approaches for both scale-out and scale-up configurations. We discuss the design and implementation of an *embedding + All-to-All* fused operator to demonstrate our scale-out approach, while we use a *GEMV + AllReduce* operator as an example of our scale-up approach which avoids intermediate buffering and copy operations.

A. Scale-out: Fused Embedding + All-to-All Operator

We develop the fused embedding + All-to-All operator as a persistent GPU HIP [10] kernel, which performs both embedding pooling (reduction-like) computations and All-to-All communication. We use the ROC_SHMEM [24] library to issue intra-kernel communication and Figure 6 shows its execution. It illustrates a two-node system where embedding tables are distributed in a model parallel fashion such that there are four tables per node. The All-to-All output and send buffers are allocated within each GPU's symmetric heap (using the *roc_shmem_malloc()* API). Memory allocated within the symmetric heap are registered with the NICs, thus allowing the NICs to directly move data between these GPU buffers. We implemented our fused *embedding + All-to-All* operator as a persistent thread kernel [36], [46] which multiplexes multiple logical embedding pooling operations onto the long-running persistent WGs. This allows us to schedule operations computing the same slice (output fragment) together (similar to the approach used in [46]) and further reduces the number of kernel invocations. The kernel is launched with a fixed, input-independent grid size (less than or equal to maximum occupancy as determined from the HIP occupancy API [19]). Each persistent WG then executes a task loop where every iteration corresponds to the computation performed by a logical WG of the original embedding kernel (*EmbeddingBag_updateOutputKernel_sum_mean*).

Figure 6a shows the logical WGs and the colors indicate which tables are processed by the WGs. Our fused kernel takes in categorical inputs and embedding tables as arguments. The illustrated example assumes a global batch size of four,



(a) Example embedding + All-to-All operation.

(b) Execution timeline of WGs for Embedding + All-to-All.

Fig. 6: Embedding + All-to-All fused operator.

the slice size to be two embedding output vectors, and that two WGs compute one slice. The pooled output embedding vectors from each table are shuffled equally across both nodes in accordance with All-to-All functionality, with the first half of the global batch stored in node 0 and the second half in node 1. Depending on the computed embedding slice, the WGs may store their results locally or communicate them to the remote node. WGs can determine the index of the computed slice using the output embedding entry and the size of the slice. The slice index along with the global batch size and node count can then be used to determine if the slice needs to be remotely communicated. For example in Figure 6a, logical WGs *WG 00 - WG 13* compute the output corresponding to the first half of global batch, while *WG 20 - WG 33* compute the second half. For node 0, *WG 00 - WG 13* store their results locally, and *WG 20 - WG 33* communicate their results to node 1. While for node 1, *WG 00 - WG 13* communicate their results to node 0 and *WG 20 - WG 33* store their results locally.

As described earlier, our approach performs communication at a granularity of a *slice*. The size of each slice is set to match the output computed by one or more WGs. In the example shown in Figure 6a, *WG 20* and *WG 30* compute slice A on node 0. While there can be multiple WGs responsible for computing a single slice, the remote communication to move the slice is initiated by the last WG that finishes the slice. In the example shown, assume *WG 30* finishes its computation for slice A after *WG 20*. Then, a thread from *WG 30* issues remote communication (remote PUT) to move slice A to node 1. We track the completion of logical WGs corresponding to every slice to identify the last completing WG and trigger communication accordingly. Implementing All-to-All using point-to-point transactions allows the communicating WGs to move the slice data to the destination in the layout required by any subsequent kernel (e.g., the interaction operation in DLRM). The output data can be shuffled at a slice granularity without requiring explicit shuffling or rearrangement. In our approach, we generate output with the shape: {local batch size, (numTables $\times$ embedding dimension)}, which can then be passed to the *interaction* operator.

Book-keeping Flags: In our approach, we maintain two sets of flags (not shown in Figure 6a) per GPU to synchronize WG communication and determine the end of communication. First, we maintain a *WG_Done* bitmask per slice where each bit indicates the completion status of the logical WGs. This bitmask is used to identify the last finishing WG responsible for issuing remote communication. Second, we maintain a *sliceRdy* flag (for both locally computed and remotely received slices) to determine when all the individual slices have been received or computed. The *sliceRdy* flags are set by the GPUs computing and communicating the slices and thus are allocated in symmetric memory. The receiving GPUs poll on the *sliceRdy* flags to determine if the slices are ready for consumption.

Communication-aware Scheduling: Figure 6b illustrates the execution timeline of the persistent WGs considering two persistent WGs per node. Each persistent WG iteratively performs the work of 16 logical WGs. In our remote communication aware approach, the logical WGs computing slices for remote communication are executed before the logical WGs computing locally consumed slices, thereby maximizing the opportunity to overlap remote communication. For example, the persistent WGs in node 0 execute logical *WG 20* and *WG 30* ahead of the logical *WG 00* and *WG 10* so that the remote communication can be overlapped with the local slice computation.

Synchronization: Upon completion of each logical WG (an iteration within the task loop), a leader thread sets the corresponding bit in the *WG_Done* bitmask as shown in the timeline. The WG also checks if it is the last one to complete by testing if the other bits in the bitmask are set. This design allows the WGs to make forward progress after setting their respective *WG_Done* flags instead of waiting on an inter-WG barrier. The last completing WG issues two remote PUT calls separated by a remote fence. The first call is to move the slice data, while the second sets its corresponding *sliceRdy* flag on the remote node. The remote fence ensures that the *sliceRdy* flag is only set after the prior PUT has completed. After executing all the logical WGs, the persistent WGs poll on

a distinct subset of *sliceRdy* flags before exiting. This ensures that the data from all slices are ready for subsequent kernels while incurring less overhead than having all WGs poll on the entire set of *sliceRdy* flags. Alternatively, synchronization can be performed using a *quiet()* call along with a barrier per GPU.

B. Scale-up: GEMV + AllReduce

Unlike remote GPUs (scale-out) which require RDMA network transactions to communicate, scale-up communication can be performed using native GPU load/store instructions over Infinity Fabric™ [4] or NVLink [17] connections. We exploit this feature to further optimize our fused kernel for scale-up communication. We create an optimized zero-copy fused kernel, where in-addition to overlapping dependent communication and computation, we eliminate the stores to intermediate buffers by directly writing the output to the peer GPU memory. The remainder of this section describes the fused *GEMV + AllReduce* operator and contrasts this scale-up approach to our previously described scale-out approach.

Figure 7 illustrates our proposed fused *GEMV + AllReduce* operator. It assumes two GPUs connected via Infinity Fabric™ within a single node and each GPU uses a temporary buffer to perform the reduction. We implement our fused operator as a persistent kernel where each persistent WG iteratively executes multiple logical WGs responsible for computing individual output tiles. The same persistent WG id across the GPUs is responsible for computing the same relative output tile to simplify dependency during reduction. In our approach, we use a two-phase direct algorithm for AllReduce because it has the least number of steps and achieves low latency for fully connected GPUs [33]. Every GPU is responsible for reducing $\frac{1}{\#GPUs}$ number of tiles (reduce scatter phase) and broadcasting their results to other GPUs (AllGather phase) as shown in Figure 7a. In this example, each GPU will compute the entire GEMV output vector but one half of the tiles will be reduced locally while the other will be communicated to the peer GPU for the reduction. For example, GPU 0 will reduce the first two tiles, while GPU 1 will reduce the last two.

We perform communication-aware scheduling of logical WGs similar to our scale-out approach, where logical WGs computing tiles which need to be communicated to other GPUs are executed ahead of the logical WGs generating locally consumed tiles. In the example timeline shown in Figure 7b, we can see that WGs in GPU 0 compute and communicate tiles $\frac{k}{2} : k-1$ (k=total output tiles) ahead of tiles $0 : \frac{k}{2}-1$ which are locally reduced. The scale-up communication is performed using GPU stores over Infinity Fabric™ , so unlike our scale-out approach where a single thread initiates the network communication, all the threads directly store their results into the destination GPU tensor, eliminating any local copy and intermediate buffering. In order to reduce the amount of synchronization between GPUs, each persistent WG only sets one ready flag per peer GPU indicating all of its associated tiles have been communicated. Once the persistent WGs have computed their local tiles, they wait for the flags to be set by their counterpart persistent WGs in the peer GPUs. Once the flags are set, they reduce the tiles and broadcast their output to the other peer GPUs.

Our scale-up fused *embedding + All-to-All* and *GEMM + All-to-All* operators also have similar implementations as explained above except no reduction is performed as the operations are fused with the All-to-All collective.

C. Overheads

Having GPU threads initiate communication within compute kernels consumes extra GPU resources which can negatively impact performance. This subsection describes those overheads.

API Latency: An obvious overhead of having GPU threads initiate networking transactions is the latency of the APIs. The impact of this overhead is limited as the communication is only triggered once per slice. However, there are other book-keeping operations (e.g., setting *WG_Done* flags) which need to be performed and add overhead. Our scale-up fused implementations use direct GPU stores for communication and subsequently do not incur any device API overhead.

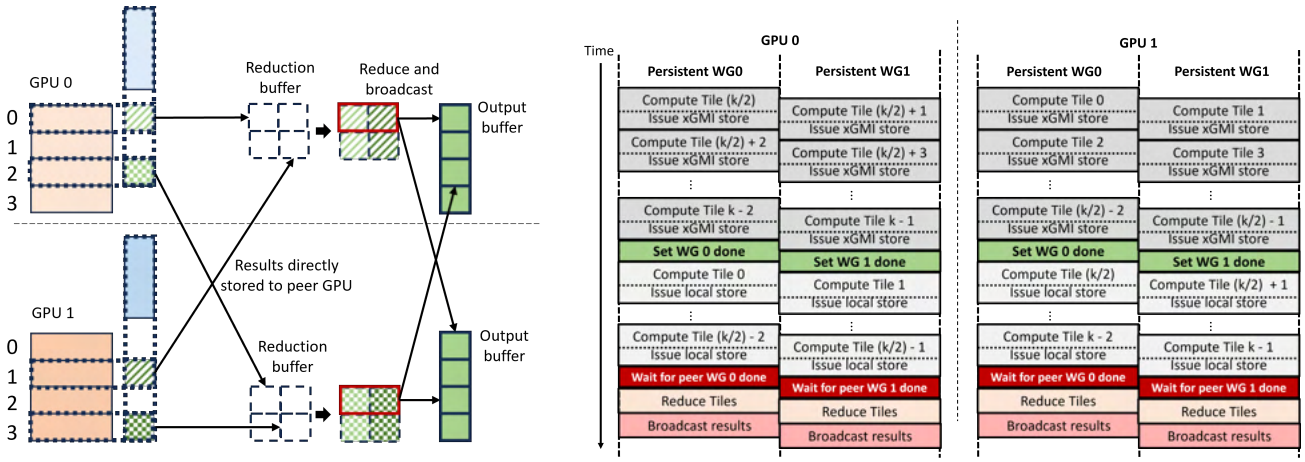
Occupancy: Invoking ROC_SHMEM API calls consume GPU registers which otherwise would have been used by the application. Reduced register availability can result in lower occupancy or increased register spilling, impacting performance. Our fused *embedding + All-to-All* kernel achieves 12.5% lower occupancy compared to the original embedding pooling kernel. Despite lower occupancy, our approach achieves better performance as shown in Section IV.

Inter-WG Synchronization: In our approach, communication is triggered once for every slice of data. Since multiple WGs could be computing an individual slice, inter-WG synchronization is required. Our implementation does not use inter-WG barrier instructions, but rather uses cross-lane operations [3] to reduce the *WG_Done* bitmask and ensure only the last completing WG per slice issues the remote communication.

D. Integration with ML Frameworks

While our approach uses the multi-threaded nature of GPU execution to perform fine-grain overlap of collective communication with computation that would otherwise not be possible, it has to be integrated within existing ML frameworks to be considered a pragmatic solution for wider adoption. In this work, we have implemented two integration approaches and this section describes how these approaches minimize programmers/developers effort.

PyTorch Integration: Our fused *embedding + All-to-All* and *GEMV + AllReduce* operators are implemented in HIP [10] and have been integrated within PyTorch. Specifically, we have created new APIs in PyTorch for 1) allocating device memory in the symmetric heap and moving a tensor from the CPU's host memory to the allocated device memory (similar to the existing $torch.tensor.to()$ API) and 2) launching fused kernel operations (e.g., $torch.embeddingAll2AllOp()$). The interfaces reduce programmer overhead as the underlying implementation is hidden from the user. Furthermore, their



(a) Example GEMV + AllReduce operation.

(b) Execution timeline of WGs for GEMV + AllReduce.

Fig. 7: GEMV + AllReduce Operator.

use can be automated using existing graph transformation optimizations within ML frameworks [27], [31].

Extending Triton Framework: Effortless development is vital for our approach to be widely adopted. We make a crucial step in this direction by extending the Triton [57] framework to include communication primitives. We chose Triton because it offers a Python-like language which is widely used by ML developers [28]. In addition, Triton is already integrated with PyTorch [1] allowing code written in Triton to be executed from PyTorch. Triton's Python-like interface enables low-overhead development of new operators. We extended it by creating a Python wrapper for the ROC_SHMEM library's scale-up communication APIs and used it to develop our *GEMM + All-to-All* fused operator (our extensions currently only support scale-up communication). It is possible to automate the generation of fused kernels using compilers that can track WG dependencies across GPU kernels [43] and expose communication operations at an intermediate representation level [12]. However, implementing such a compiler flow is beyond the scope of this work.

E. Hardware features required

Our approach does not require any hardware changes, however it capitalizes on recent GPU features to minimize implementation overheads. We discuss these features and their alternatives in this section.

Peer memory access using direct LD/ST: Our zero-copy optimizations for scale-up fused operators rely on GPUs being able to access peer GPU memory using direct load/store instructions. GPU vendors already offer proprietary high-speed interconnect solutions (e.g., NVLink, Infinity Fabric™) with such support. Further, open standard interconnects like *Ultra Accelerator Link* [30] are also being developed with similar support.

Inter-WG synchronizations: Depending on the slice size, our approach may require inter-WG synchronization. Our implementation uses cross-lane instructions [3] to reduce ready flags and detect when all WGs are done. Barrier or atomic instructions can also be used to the same effect but with a larger overhead. Dedicated hardware features, such as threadblock cluster barriers and distributed shared memory [16], may also be beneficial.

GPU cache flush instructions: GPU cache flush instructions allow output computed by WGs to be visible to the NIC without requiring the kernel to complete or needing the ROC_SHMEM symmetric heap to marked uncacheable.

TABLE I: System Setup.

GPU	AMD Instinct™ MI210
Software	PyTorch v2.0, ROCm v5.4, ROC_SHMEM v1.6
Scale-up	4 GPUs fully connected over Infinity Fabric™ (80GB/s)
Scale-out	2 nodes (with x1 GPU), connected over IB (20 GB/s)

TABLE II: Scale-out simulation setup.

DLRM Model Parameters [48]	
Embedding dimension	92
MLP layer	Avg. size: 682, # layers: 43
Avg. pooling size	70
ASTRA-Sim Network Parameters [54]	
Topology	2D Torus (BW: 200Gb/s, lat: 700ns)

IV. Evaluation

We evaluated our scale-up fused operators on four GPUs connected via Infinity Fabric™ , while our scale-out operators are evaluated using both real hardware (two GPU nodes connected via Infiniband) and simulation (128 GPU nodes).

A. Setup

We compared our fused operators against bulk-synchronous, sequential execution of computation kernels followed by RCCL [22] collective communication kernels. Sequential execution is the out-of-the-box behavior for these kernels because their data dependencies prevent them from being executed concurrently. Table I shows the system setup used for our evaluations.

Embedding + All-to-All: We evaluated fusing embedding operations (embedding dim=256 [48]), as implemented in the

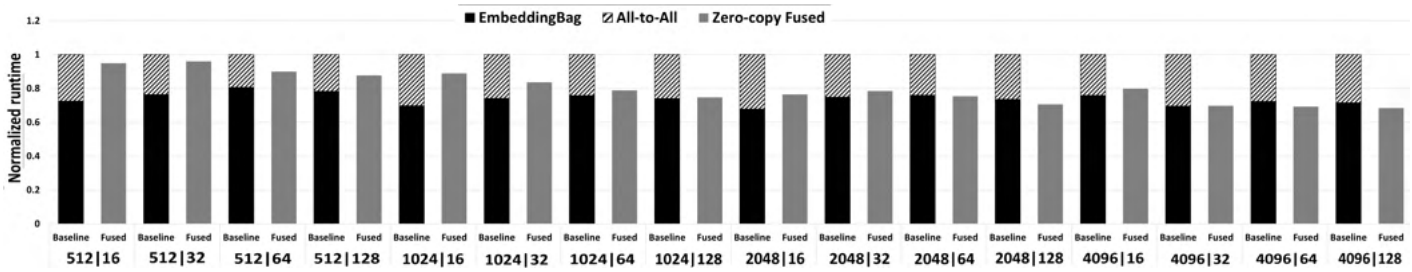


Fig. 8: Normalized execution time: Embedding Bag + All-to-All.

public DLRM [7] code, with All-to-All. In our fused kernel, a cluster of 32 logical WGs compute an output slice to achieve a much higher degree of fine-grain communication-computation overlap than what would be possible by splitting the two large kernels into many smaller kernels. For example, a batch size=2048 and tables/GPU=256 configuration results in 512k total logical WGs and requires 16384 kernel launches, which is such a significant overhead [45], [48], [61] that we did not evaluate it.

We varied the global batch-size and the number of embedding tables processed per GPU across different inter-node and intra-node configurations. Each configuration is labeled as: {global batch size | embedding tables per GPU} in the graphs. For inputs, we used the data generator in DLRM. For large scale-out simulation, we evaluated the entire DLRM [7] application in ASTRA-Sim [53]. We modified ASTRA-Sim's execution graph to model our fused *embedding + All-to-All* kernel. The DLRM model and simulator parameters are shown in Table II. The per-kernel execution times used in ASTRA-Sim were collected on an AMD Instinct™ MI210 GPU using the ROC-profiler [21].

GEMV + AllReduce: This operator is specific to inference, so we evaluate them in a scale-up configuration. The computation involved is relatively small (inference workload) and thus splitting them into smaller kernels to achieve computation-communication overlap would under-utilize GPUs and exacerbate kernel launch overheads. Thus, we compare our approach against a bulk-synchronous RCCL-based baseline. We select the input matrix and vector dimensions to reflect the sizes in current and future transformer models [51].

GEMM + All-to-All: The evaluated fused operator is implemented in Triton. We use the publicly available Triton GEMM implementation [23], [29] and modify it to perform All-to-All using our communication extensions. The input matrix sizes are based on commonly seen dimensions in MoE layers [40].

B. Scale-up evaluation

We evaluated the benefits of the zero-copy, fused kernel on a single node with four GPUs. All communication is performed at the GPU thread granularity (not slice) using stores across the Infinity Fabric™ links.

Embedding + All-to-All: Figure 8 shows the execution time of our approach normalized against the time taken by the bulk-synchronous baseline. Overall, our approach achieves on average 20% (and up to 32%) lower execution time. For larger batches, we observe that our fused implementation even outperforms the baseline embedding bag. This is because, as part of our zero-copy optimizations, the fraction of embedding bag output to be communicated is directly stored (over Infinity Fabric™) into peer GPUs, reducing the amount of data written into local HBM by each GPU. This reduces the local write contention (especially for configurations with larger batch size and higher embedding table count) resulting in better performance than baseline.

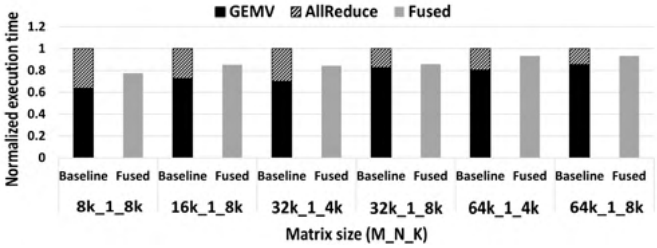


Fig. 9: Normalized execution time: GEMV + AllReduce.

GEMV + AllReduce: Figure 9 shows the execution time of our *GEMV + AllReduce* fused operator normalized against the time taken by the bulk-synchronous execution of the GEMV kernel followed by an intra-node AllReduce kernel. Our approach achieves on average 13% (and up to 22%) lower execution time. The performance benefits is less for the larger inputs (M = 64k) because as the output vector size grows, GEMV further dominates the execution time and the contention for the Infinity Fabric™ links increases.

GEMM + All-to-All: Figure 10 shows the execution time of our *GEMM + All-to-All* fused operator normalized against time taken by the bulk-synchronous execution of the GEMM kernel followed by an intra-node All-to-All kernel. Our approach lowers execution time by 12% on average and up to 20%. Since we are using a generic GEMM implementation provided with Triton, the GEMM dominates the overall execution time and limits the benefits of our approach.

C. Inter-node evaluation

WG Profiling: We first demonstrate the effectiveness of overlapping embedding operations with All-to-All communication. We profile the persistent WGs' execution timeline for a configuration with batch size = 2048, tables/GPU = 256, and with slice size such that each slice is computed by 16 WGs. WGs computing the same slice (cluster of 16) are sorted based on their time to complete embedding pooling operations

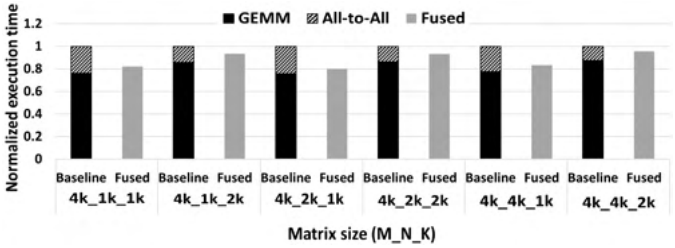


Fig. 10: Normalized execution time: GEMM + All-to-All.

(earliest first). Figure 11 illustrates the execution timelines for the first 32 persistent WGs along with the points in time when non-blocking network transactions were issued. Figure 11 also shows the instances when the locally consumed slices are computed (marked as *Local slice completion*) to illustrate the effect of communication-aware scheduling. We can see that some persistent WGs are issuing remote communication while others are performing the computation, demonstrating our approach achieves fine-grained communication-computation overlap. Furthermore, asynchronous network transactions allow the communicating WGs (e.g., WG 15 and 31) to continue performing embedding pooling without being blocked. Figure 11 shows that the last WGs (WG 15 and WG 31) within each cluster of 16 WGs issues most of the communication. This is because each persistent WG iteratively computes individual slices. The last completing WG for one slice often stays the last across multiple slices (recall that only the last completing WG issues the remote non-blocking PUT). Additionally, we can see that the remote communication calls are issued before computing the local slices, maximizing the opportunity to hide remote communication. The time spent waiting on data differs across WGs because each WG waits for a distinct subset of the *sliceRdy* flags.

Execution time: Figure 12 shows the time taken by our fused embedding + All-to-All kernel normalized to the baseline executing separate embedding operation and All-to-All collective kernels. Our fused kernel uses a slice size of 32 embeddings and Figure 12 demonstrates that our approach

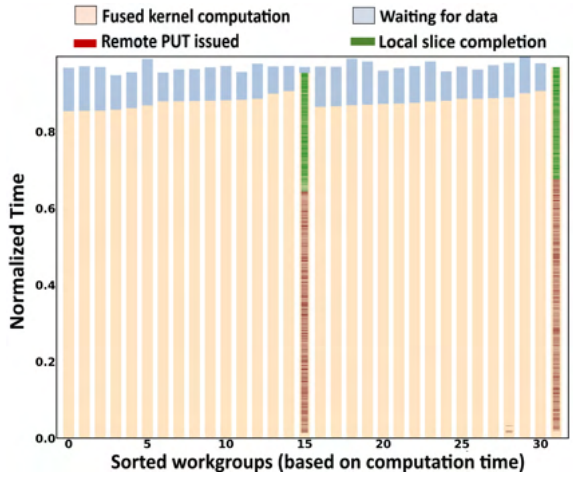


Fig. 11: Profiled timeline of persistent WG.

benefits a wide set of batch sizes and embedding table counts, achieving 31% average reduction in execution time (up to 58%). For smaller global batch sizes (e.g., 256), we note that the performance benefit is more than what could be achieved by fully overlapping communication. This is because smaller batch sizes result in poor compute utilization for the baseline, while our approach achieves much higher compute utilization by processing all tables within a single fused kernel.

Occupancy effect: Our fused kernel implementation results in 12.5% lower GPU occupancy than the baseline due to the extra registers required by GPU-initiated networking operations. However, this loss of occupancy does not degrade performance. Figure 13 shows the variation in the execution time (global batch size: 1024, tables/GPU: 256) across different GPU occupancy. The figure only shows results up to 87.5% occupancy because that was the maximum achievable occupancy relative to baseline. We see that as the occupancy is increased from 25% to 75%, the parallelism increases and consequently the execution time reduces by 46%. However, increasing the occupancy further from 75% to 87.5% results in the execution time increasing by 25%. At this higher occupancy level, the memory intensive embedding operations encounter significant memory contention, exemplifying the trade-off between parallelism and memory contention.

Communication-aware Scheduling: In our approach, we implement communication-aware WG scheduling to maximize the opportunity to hide remote communication. Figure 14 shows the fused *embedding + All-to-All* kernel execution time of both nodes (normalized to baseline:node 0) with and without communication-aware scheduling. The baseline communication-oblivious scheduling starts from WG (0,0,0) and then proceeds sequentially. In baseline scheduling, node 0 and 1 have an average execution time skew of $\sim$7% while using communication-aware scheduling exhibits only a $\sim$1% average execution time skew. Many distributed ML models, including DLRM, periodically synchronize across nodes (e.g., during synchronous gradient descent), and thus execution skew can reduce the overall performance. The higher skew in the baseline is due to the higher execution time for node 1. As part of our fused kernel implementation, WGs in node 1 will wait for slices from node 0 and vice-versa. Node 1 has the same logical WG schedule under both strategies, where it first computes the slices to be communicated to node 0 (please see Figure 6b). However, with communication-oblivious scheduling, node 0 computes the slices to be remotely communicated after the locally consumed slices. Thus the remote communication is not hidden behind local slice computation, delaying the completion of node 1.

D. Large scale-out evaluation

We evaluated the benefits of our approach on DLRM training distributed across 128 nodes (with one GPU each) using ASTRA-Sim. Figure 15 shows the normalized execution time for performing one DLRM training pass. The embedding operations in both the forward and backward passes are overlapped with their dependent All-to-All collective operations

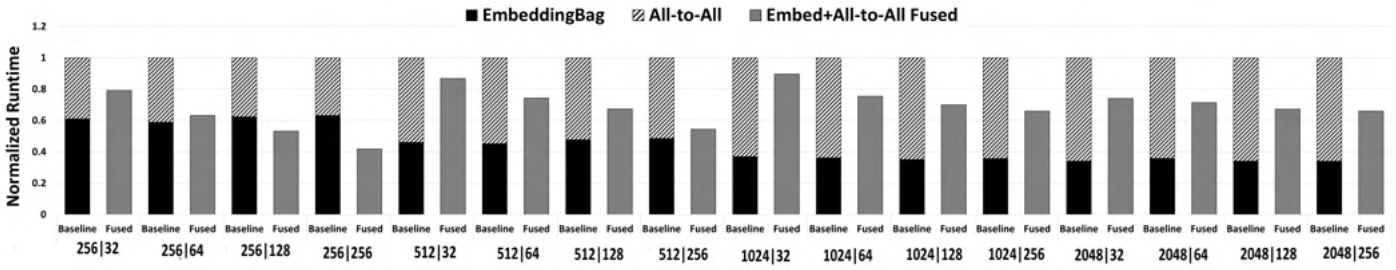


Fig. 12: Normalized execution time (scale-out): Embedding Bag + All-to-All.

using WG-level parallelism. We can see that our approach is able to hide most of embedding operations, achieving on average a ∼21% reduction in the overall execution time.

V. Related Work

In this section, we discuss prior research done on optimizing distributed ML models, overlapping computation with CPU-initiated inter-node and DMA-assisted intra-node communication. We also discuss previous work on GPU-initiated communication.

T3 [51] proposes hardware track and trigger mechanisms to detect GPU stores and initiate intra-node DMA communication to overlap it with computation. While this work aims to achieve communication-computation overlap, it requires hardware modifications to track GPU stores using a new widget invisible to programmers that can be difficult to debug. In contrast, our approach requires no hardware changes and has been implemented in commercially available frameworks.

Wang *et al*. [59] propose decomposing the original collective communication and its dependent computation into smaller pieces. The communication of an individual data shard can be overlapped with the computation of another. Our approach is conceptually similar to this as both aim at overlapping communication of a portion of data with the computation of another. However, unlike us, Wang *et al*.'s approach will result in a higher number of kernel invocations where each sharded kernel is smaller than the original one. For their use case [59], the sharded kernels are sufficiently large to amortize the kernel invocation overhead but this is not always the case [48]. Further, their approach requires additional operations to combine the individual partial results. Jangda *et al*. [42] also proposes similar optimizations but they use a new domain-specific language to express ML programs in the form of computation and communication operations. Our approach instead uses GPU-initiated networking integrated within existing ML frameworks, to hide communication without requiring additional kernel launches, and computations.

Mudigere *et al*. [48] proposed Neo system to improve distributed DLRM execution. This system employs 4D parallelism strategy (table-wise, row-wise, column-wise, and data parallelism) for distributing embedding computations evenly across GPUs. Furthermore, they fuse compute kernels to minimize kernel-launch overheads and memory requirements, and employ software-managed caching to leverage hardware memory hierarchy. However, in this paper, we try to address the large All-to-All communication latency exposed in distributed DLRM training by fusing and overlapping All-to-All with embedding operations.

Wang *et al*. [60] propose using GPU-initiated networking to hide remote access latency in GNNs. While this work uses GPU-initiated communication, it focuses on optimizing irregular communication by interleaving local neighbor aggregation with remote neighbour access. Their work targets a different problem than that addressed in this paper and requires a completely different optimizations and implementation.

Unlike our approach which aims at hiding the collective communication latency, there have been efforts to reduce them. Cai *et al*. [32] have proposed an approach to synthesize collective algorithms tailored for a specific hardware topology. ARK [41] proposes offloading intra-node communication to DMA driven by GPU to reduce control-plane overhead and data-plane interference. Others [39], [54] have proposed optimal communication schedule for specific collectives aimed at reducing contention and improving link utilization. These optimizations are orthogonal to our approach.

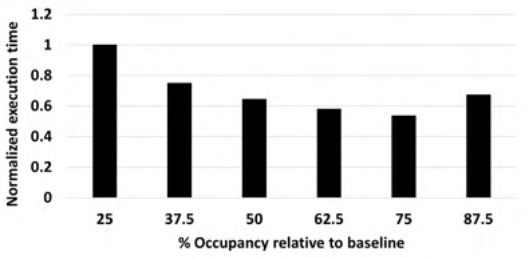


Fig. 13: Impact of WG occupancy on execution time.

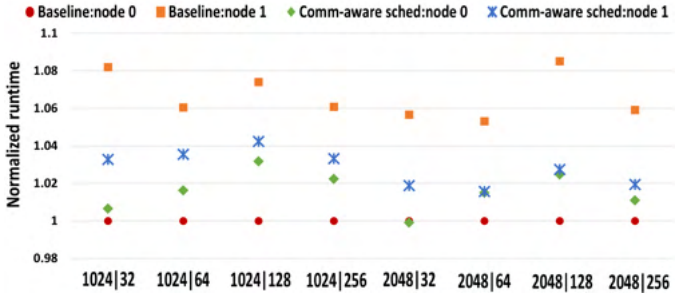


Fig. 14: Impact of communication-aware WG scheduling.

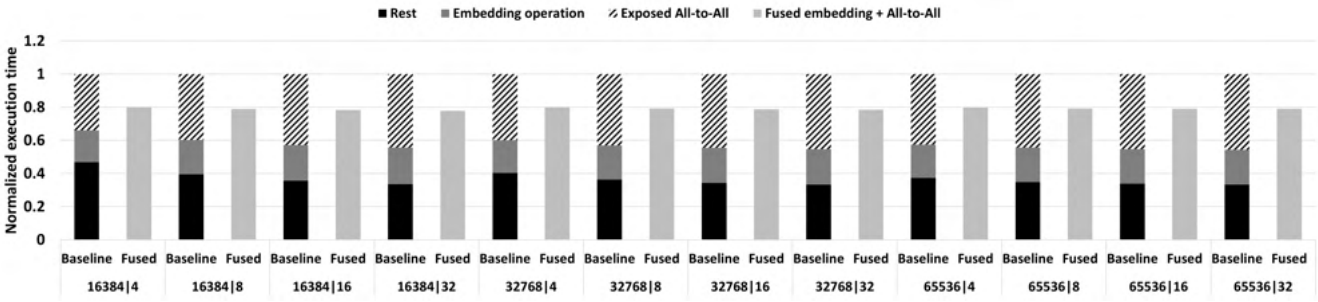


Fig. 15: Scale-out simulation normalized execution time.

VI. Conclusion

In this paper, we hide the collective communication with dependent computation at a fine granularity. GPU-initiated intra-kernel communication enables wavefront/warp-, WG- or WG-cluster granular communication unlike CPU-initiated networking which provides only kernel-granular communication. We developed prototype fused *embedding + All-to-All*, *GEMV + AllReduce* and *GEMM + All-to-All* kernels where fragments of computed outputs are communicated in parallel to other WGs performing the remainder of the computation. We propose remote communication-aware scheduling where the logical WGs computing the remote slices are executed before the WGs computing the local slices, maximizing the opportunity to overlap remote communication. We further optimize scale-up communication among GPUs by developing zero-copy fused kernels. Here, GPU threads perform the computation and store the results directly to the destination address at the peer GPU, thereby eliminating intermediate stores to local memory. We expose our fused operators as new PyTorch operators as well as develop fused operators (*GEMM + All-to-All*) using our Triton framework extensions which include communication primitives to demonstrate integration with ML frameworks.

We evaluated our approach both on actual hardware and using simulation. Our scale-up evaluations show that zero-copy fused *embedding + All-to-All* kernel on average achieves 20% (and up to 32%) lower execution time than the baseline. Our *GEMM + All-to-All* and *GEMV + AllReduce* evaluations show that our approach achieves 13% (up to 22%) and 12% (up to 20%) lower execution time respectively. Our inter-node evaluations show that the execution time taken by fused kernel *embedding + All-to-All* is on average 31% (and up to 58%) lower than that of baseline embedding operations and All-to-All collective. We evaluated the impact of proposed communication-aware WG scheduling and show that it achieves 6% lower execution skew than communication-oblivious scheduling. Finally, we used ASTRA-Sim to perform large scale-out simulations to evaluate the benefits of our approach on the entire DLRM training run. Our evaluations show that using fused embedding + All-to-All kernels reduce the training time by 21% for a 128 node system.

As part of our future work, we plan extend Triton support to include scale-out communication and compare our approach against manually splitting the computation and communication operations into smaller kernels and overlapping them using streams.

VII. Acknowledgements

We would like to thank our reviewers for their valuable feedback which helped improve the paper. We would also like to thank Ralph Wittig (AMD Research and Advanced Development) for supporting this work, and Simon Waters, Jason Furmanek (Triton team, AMD) for their help and support. AMD, the AMD Arrow logo, Instinct, Infinity Fabric, RCCL, ROCm, ROC_SHMEM and combinations thereof are trademarks of Advanced Micro Devices, Inc. PyTorch, the PyTorch logo and any related marks are trademarks of The Linux Foundation. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] Accelerating Triton Dequantization Kernels for GPTQ. https://pytorch.org/blog/accelerating-triton/.
[2] AMD CDNA™ 2 ARCHITECTURE. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf.
[3] AMD GCN Assembly: Cross-Lane Operations. https://gpuopen.com/learn/amd-gcn-assembly-cross-lane-operations/.
[4] AMD Infinity Architecture. https://www.amd.com/en/technologies/infinity-architecture.
[5] "AMD Instinct MI200" Instruction Set Architecture. https://www.amd.com/system/files/TechDocs/instinct-mi200-cdna2-instruction-set-architecture.pdf.
[6] An Introduction to CUDA-Aware MPI. https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.
[7] Deep Learning Recommendation Model for Personalization and Recommendation Systems. https://github.com/facebookresearch/dlrm.
[8] Fully Sharded Data Parallel: faster AI training with fewer GPUs. https://engineering.fb.com/2021/07/15/open-source/fsdp/.
[9] GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html#.
[10] HIP: Heterogenous-computing Interface for Portability. https://rocm-developer-tools.github.io/HIP/.
[11] ML Parallelism. https://huggingface.co/docs/transformers/v4.15.0/parallelism.
[12] MPI Dialect. https://discourse.llvm.org/t/rfc-mpi-dialect/74705.
[13] MSCCL++. https://github.com/microsoft/mscclpp.
[14] NCCL: Optimized primitives for inter-gpu communication. https://github.com/NVIDIA/nccl.
[15] NVIDIA Grace Hopper Superchip Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/.
[16] NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.
[17] NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.

[18] NVSHMEM. https://developer.nvidia.com/nvshmem.

[19] OccupancyHIP API. https://docs.amd.com/bundle/HIP-API-Guide-v5.2/page/group___occupancy.html#ga322e4690ca20dbf8a07293f2a1105c94.

[20] Pytorch. https://pytorch.org/.

[21] ROC-profiler. https://github.com/ROCm-Developer-Tools/rocprofiler.

[22] ROCm Communication Collectives Library. https://github.com/ROCmSoftwarePlatform/rccl.

[23] ROCm/triton. https://github.com/ROCm/triton.

[24] ROC_SHMEM. https://github.com/ROCm-Developer-Tools/ROC_SHMEM.

[25] Tensile. https://github.com/ROCmSoftwarePlatform/Tensile.

[26] The EPYC™ CPU and INSTINCT™ MI250X GPUs in Frontier. https://olcf.ornl.gov/wp-content/uploads/2-15-23-AMD-CPU-GPU-Frontier-Public.pdf.

[27] TORCHSCRIPT. https://pytorch.org/docs/stable/jit.html.

[28] Triton. https://openai.com/research/triton.

[29] triton. https://github.com/openai/triton/tree/main.

[30] Ultra Accelerator Link is an open-standard interconnect for AI accelerators being developed by AMD, Broadcom, Intel, Google, Microsoft, others. https://www.tomshardware.com/tech-industry/artificial-intelligence/amd-broadcom-intel-google-microsoft-and-others-team-up-for-ultra-accelerator-link-an-open-standard-interconnect-for-ai-accelerators.

[31] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

[32] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[33] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K. (D K) Panda. Nv-group: link-efficient reduction for distributed deep learning on modern dense gpu systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.

[34] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[35] Huansong Fu, Manjunath Gorentla Venkata, Ahana Roy Choudhury, Neena Imam, and Weikuan Yu. High-performance key-value store on openshmem. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[36] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, 2012.

[37] Khaled Hamidouche, Ammar Ahmad Awan, Akshay Venkatesh, and Dhabaleswar K. Panda. Cuda m3: Designing efficient cuda managed memory-aware mpi by exploiting gdr and ipc. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, 2016.

[38] Khaled Hamidouche and Michael LeBeane. GPU Initiated OpenSHMEM: Correct and Efficient Intra-Kernel Networking for DGPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 336–347, 2020.

[39] Jiayi Huang, Pritam Majumder, Sungkeun Kim, Abdullah Muzahid, Ki Hwan Yum, and Eun Jung Kim. Communication algorithm-architecture co-design for distributed deep learning. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021.

[40] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale, 2022.

[41] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. Ark: Gpu-driven code execution for distributed deep learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[42] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 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*, 2022.

[43] Abhinav Jangda, Saeed Maleki, Maryam Mehri Dehnavi, Madan Musuvathi, and Olli Saarikivi. A framework for fine-grained synchronization of dependent gpu kernels, 2023.

[44] Kawthar Shafie Khorassani, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K. Panda. Performance evaluation of mpi libraries on gpu-enabled openpower architectures: Early experiences. In *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers*, 2019.

[45] Sumin Kim, Seunghwan Oh, and Youngmin Yi. Minimizing gpu kernel launch overhead in deep learning inference on mobile gpus. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, 2021.

[46] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-aware cta clustering for modern gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[47] Patrick MacArthur and Robert D. Russell. A performance study to guide rdma programming decisions. In *2012 IEEE 14th International Conference on High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012.

[48] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[49] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. 2019.

[50] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *ArXiv*, 2023.

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

[52] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022.

[53] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 22-26, 2020*, 2020.

[54] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. Themis: A network bandwidth-aware collective scheduling policy for distributed training of dl models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[55] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[57] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In

Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2019.

[58] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap, 2022.

[59] 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. 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*, 2022.

[60] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. Empowering gnns with fine-grained communication-computation pipelining on multi-gpu platforms, 2022.

[61] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An efficient gpu embedding cache for personalized recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 We propose a novel approach to fuse computation and collective communication within the same kernel, thus overlapping collective communication, and reducing peak bandwidth. Further, we propose zero-copy fused kernels where the results are directly written to the peer GPU memory thereby eliminating intermediate buffering and copy operations.
- C_2 We develop three first of their kind fused communication-computation prototype kernels to address the collective overhead in DLRM, Transformers, and MoE architectures. Specifically, we develop *embedding pooling + All-to-All*, *GEMV + AllReduce*, and *GEMM + All-to-All* fused operators and evaluate them against RCCL-based bulk-synchronous approaches.
- C_3 We implement two different approaches to integrate fused operators within ML frameworks. 1) We expose fused communication-computation kernel as a new operator within PyTorch to be transparently used by developers. 2) We extend Triton framework to include communication primitives allowing users to develop custom fused kernels catering to their needs. In this work, we use our Triton extensions to develop *GEMM + All-to-All* kernel.

B. Computational Artifacts

This work was done as part of AMD's research and advanced development efforts. Currently, we cannot release the implementations of the prototype fused operators due to AMD-internal legal review requirements and the confidential nature of some of the components. However, as part of this appendix we detail the individual software packages, and communication runtime used to develop and execute the approaches presented in the paper.

- A_1 Fused *embedding pooling + All-to-All*, *GEMV + AllReduce* implementation
- A_2 Fused *GEMM + All-to-All* Triton implementation
- A_3 AstraSim DLRM model with Fused *embedding pooling + All-to-All* operator

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Figure 8-9, 11-14
A_2	C_1, C_2, C_3	Figure 10
A_3	C_2	Figure 15

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This is the implementation of fused *embedding pooling + All-to-All* and *GEMV + AllReduce* kernels explained in Section 3 of the paper. This artifact applies to all three contributions of our paper (C_1, C_2, and C_3). These fused operators illustrate the fine-grained overlap of communication and computation achieved using our approach and are integrated within PyTorch as a new operator.

Expected Results

Embedding pooling + All-to-All: This kernel performs both embedding bag pooling and All-to-All collective as part of its execution. The kernel takes in embedding bag tables as inputs along with their corresponding categorical inputs. It then performs the pooling operation on the table embedding vectors based on the categorical inputs and the results are communicated to peer GPUs using ROC_SHMEM library calls. This kernel fuses All-to-All communication with embedding bag pooling within the same kernel achieving communication overlap. Our approach achieves lower execution time when compared against the time taken by bulk-synchronous embedding pooling and All-to-All in public DLRM implementation for all configurations detailed in Section 4.

GEMV + AllReduce: This kernel performs both GEMV (matrix-vector multiplication) and AllReduce collective as part of its implementation. The fused kernel multiplies the input matrix and vector in an output stationary tiled manner. The individual tiles are communicated and reduced using using ROC_SHMEM library calls as explained in Section 3. We use an internal GEMV kernel as baseline and AllReduce from RCCL. Our fused kernel should be faster than bulk-synchronous execution of baseline GEMV and AllReduce operations for all input sizes specified in Section 4.

These new fused kernels are exposed in PyTorch as new operators by extending the PyTorch ATen "native" functions as described in https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/README.md. The HIP-only implementation is used to perform profiling experiments presented in Section 4.C (Figure 11, 13, and 14). We use $s\_memrealtime$ instruction (https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/instinct-mi200-cdna2-instruction-set-architecture.pdf) to collect intra-kernel timestamps to perform WG granular profiling.

Expected Reproduction Time (in Minutes)

Setup: This artifact requires ROC_SHMEM and PyTorch to be installed. Compiling and installing ROC_SHMEM along with its dependencies will take about 1 hour to install. Installing PyTorch from source will require around 5 hours.

Execution: The expected execution time for this artifact is under 20 minutes per configuration on MI210™ GPUs.

Artifact Setup (incl. Inputs)

Hardware: 1) Scale-up configurations: 4x MI210™ GPUs fully-connected using Infinity Fabric™. 2) Scale-out configurations: 2x nodes each with 1x MI210™ GPUs connected using a InfiniBand 200 Gb/s switch. Our experiments were done with an AMD EPYC™ 7282 CPU running Ubuntu 20.04, however the specific CPU used should not impact the results.

Software: The fused kernels are implemented in HIP using ROC_SHMEMv1.6 and evaluated on ROCm v5.4 and RCCL (for ROCm 5.4). The fused operators are integrated into PyTorch 2.0. The PyTorch DLRM implementation used is available at https://github.com/facebookresearch/dlrm.

Datasets / Inputs: For *embedding pooling + All-to-All*, we used the data generator with DLRM (https://github.com/facebookresearch/dlrm) to generate input data based on configuration listed in Section 4. The input is generated as part of the PyTorch code and no dedicated data generation step is required. For evaluating the HIP-only *embedding pooling + All-to-All* operator, the data generated by DLRM data generator is logged into a separate file, which is processed by the host driver code to setup the embedding tables and categorical inputs. For *GEMV + AllReduce*, we random generated matrices and vectors of sizes detailed in Section 4. The matrix and vector values do not impact the evaluation and results.

Installation and Deployment: Instructions to install ROCm are available here: https://rocm.docs.amd.com/en/docs-5.4.3/deploy/linux/installer/install.html Instructions to install ROC_SHMEM are available here: https://github.com/ROCm/ROC_SHMEM. ROC_SHMEM has dependencies on:

- ROCm (we used v5.4)
- ROCm-aware UCX (required by MPI, we used v1.11)
- ROCm-aware MPI (we used v4.0)

We use the GPU-IB backend for inter-node communication in ROC_SHMEM (default option on installing) and set symmetric heap size to 4GB using the runtime parameter ROC_SHMEM_HEAP_SIZE. PyTorch with the new fused operators must be compiled from source (instructions: https://rocm.docs.amd.com/projects/install-on-linux/en/develop/how-to/3rd-party/pytorch-install.html). Since ROC_SHMEM uses MPI, torch.distributed module should be used with MPI backend for evaluating fused operators. Note that baseline uses RCCL backend. All the HIP code can be compiled using HIPCC (v5.4) which gets installed as part of the ROCm stack.

Artifact Execution

The work flow consists of 3 steps: 1) Install ROC_SHMEM and PyTorch modified to include fused operators, 2) Execute the baseline and fused implementations with the appropriate input arguments (batch size, embedding tables for *embedding + All-to-All* and matrix, vector dimensions for *GEMV + AllReduce*. The HIP-only fused kernel artifact is run collect the profiling results, and 3) The execution time is measured as part of the implementations using HIP events which is used to plot the graphs. We also have HIP-only implementations which we use for profiling (Figure 11, 13, and 14). For Figure 11, the artifact collects and logs intra-kernel timestamps which we use to plot the timeline using a Python script.

The parameters for both *embedding + All-to-All* and *GEMV + AllReduce* evaluation were determined based on prior research work [47], [50] and are listed in section 4. Both experiments were repeated 10 times.

Artifact Analysis (incl. Outputs)

The output of the experiments are execution times. The baseline execution time is measured separately for compute (embedding pooling, GEMV) and collective (All-to-All, AllReduce). The execution times are normalized and plotted.

B. Computational Artifact A_2

Relation To Contributions

This is the implementation of fused *GEMM + All-to-All*. This artifact demonstrates integration of our approach with ML frameworks using Triton. The Triton framework is extended to enable intra-node GPU communication. We create Python wrappers for ROC_SHMEM library APIs using pybind11 (https://github.com/pybind/pybind11) and integrate it within the Triton kernel.

Expected Results

GEMM + All-to-All: This Triton kernel performs both GEMM (matrix-matrix multiplication) and All-to-All collective in a single kernel. The fused kernel multiplies the input matrices in an output stationary tiled manner. The individual tiles are communicated and reduced using using ROC_SHMEM library calls similar to *GEMV + AllReduce*. The baseline Triton GEMM used is from https://github.com/openai/triton/blob/main/python/tutorials/03-matrix-multiplication.py. This Triton implementation is modified to perform intra-node All-to-All using stores over Infinity Fabric™. We assume top-2 routing and uniform distribution across experts for our evaluation, so each token (represented by row vector of GEMM output) is communicated to two destination GPUs. Our fused kernel should be faster than bulk-synchronous execution of baseline GEMM and All-to-All operations for all input sizes specified in Section 4 of the paper.

Expected Reproduction Time (in Minutes)

Setup: Triton setup should take less than 30 minutes.

Execution: The expected execution time for this artifact is under 20 minutes per configuration on MI210™ GPUs.

Artifact Setup (incl. Inputs)

Hardware: 4x MI210™ GPUs fully-connected using Infinity Fabric™.

Software: The fused *GEMM + All-to-All* kernel are written in Triton using ROC_SHMEMv1.6 for communication.

Datasets / Inputs: We random generated the input matrices of dimensions detailed in Section 4. The input matrix values do not impact the evaluation and results.

Installation and Deployment: Instructions to install ROCm are available here: https://rocm.docs.amd.com/en/docs-5.4.3/deploy/linux/installer/install.html Instructions to install ROC_SHMEM are available here: https://github.com/ROCm/ROC_SHMEM. ROC_SHMEM has dependencies on:

- ROCm (we used v5.4)
- ROCm-aware UCX (required by MPI, we used v1.11)
- ROCm-aware MPI (we used v4.0)

Triton can be installed using instructions provided here: https://github.com/ROCm/triton.

Artifact Execution

The work flow consists of 3 steps: 1) Install ROC_SHMEM and Triton modified to include communication extensions, 2) Execute the baseline and fused implementations with the appropriate input matrix dimensions, and 3) The execution time is measured as part of the implementations using HIP events which is used to plot the graphs.

The parameters for *GEMM + All-to-All* evaluation were determined based on prior research work [39] and are listed in section 4. Experiments were repeated 10 times.

Artifact Analysis (incl. Outputs)

The output of the experiments are execution times. The baseline execution time is measured separately for GEMM and All-to-All collective. The execution times are normalized and plotted.

C. Computational Artifact A_3

Relation To Contributions

This artifact is for modeling of *embedding + All-to-All* for large scale-out system configurations. It consists of baseline DLRM model and the model with fused embedding bag pooling and All-to-All collective operation. These models are captured in the form CHAKRA traces which are provided as an input to AstraSim simulator.

Expected Results

We model DLRM execution on 128 node system and compare the execution time of one training pass with and without using our fused *embedding + All-to-All* operator. Evaluations with fused kernel should be faster than the baseline for all input sizes specified in Section 4.

Expected Reproduction Time (in Minutes)

Setup: AstraSim installation should take less than 10 minutes.

Execution: The expected execution time for this artifact is under 30 minutes per configuration on AMD EPYC™ 7282 CPU.

Artifact Setup (incl. Inputs)

Hardware: Any x86 CPU should be able to run AstraSim simulator, we ran the simulations on AMD EPYC™ 7282 CPU.

Software: We used AstraSim 2.0 for simulating and evaluating DLRM.

Datasets / Inputs: The inputs to the simulator are CHAKRA traces generated using the trace generator available as part of CHAKRA (https://github.com/mlcommons/chakra). The model parameters used are listed in Table 2, Section 4 in the paper. The per-kernel execution times used in the traces were collected on an AMD Instinct™ MI210 GPU using ROC-profiler.

Installation and Deployment: AstraSim can be installed using the instructions here: https://github.com/astra-sim/astra-sim. The execution traces (.et files) which are passed as input to AstraSim simulator can be generated using et_generator.py generator script in CHAKRA. The generator script is modified to generate the baseline DLRM model (parameters in Table 2, Section 4 of the paper) annotated with profiled compute kernel execution times. We also generate execution traces corresponding to DLRM model with fused *embedding + All-to-All* operators by overlapping embedding bag pooling computation with All-to-All collective.

Artifact Execution

The work flow consists of 3 steps: 1) Generate the execution trace files for DLRM corresponding to the individual configurations (batch size, number of embedding tables) 2) Simulate the baseline and fused operator variant with the appropriate trace inputs, and 3) record the execution time for individual phases of DLRM execution from the output log file and plot the graphs.

The parameters for DLRM model are obtained from prior research work [47] and are listed in Table 2 in section 4 of the paper. Simulator is deterministic so executing once is sufficient.

Artifact Analysis (incl. Outputs)

The output of the simulation is an output log file containing simulated execution times. The simulated times are recorded, normalized and plotted.

Artifact Evaluation (AE)

Not provided for Artifact Description deadline.