Accelerating Multi-GPU Embedding Retrieval with PGAS-Style Communication for Deep Learning Recommendation Systems

Yuxin Chen
University of California, Davis
Davis, USA
yxxchen@ucdavis.edu

Aydın Buluç
Lawrence Berkeley National Laboratory
Berkeley, USA
abuluc@lbl.gov

Katherine Yelick
University of California, Berkeley
Berkeley, USA
yelick@berkeley.edu

John D. Owens
University of California, Davis
Davis, USA
jowens@ece.ucdavis.edu

***Abstract*—In this paper, we propose using Partitioned Global Address Space (PGAS) GPU one-sided asynchronous small messages to replace the widely used collective communication calls for sparse input multi-GPU embedding retrieval in deep learning recommendation systems. This GPU PGAS communication approach achieves (1) better communication and computation overlap, (2) smoother network usage, and (3) reduced overhead (due to the data unpack and rearrangement steps associated with collective communication calls). We implement a CUDA embedding retrieval backend for PyTorch that supports the proposed PGAS communication scheme and evaluate it on deep learning recommendation inference passes. Our backend outperforms the baseline using NCCL collective calls, achieving 1.97x speedup for the weak scaling test and 2.63x speedup for the strong scaling test in a 4 GPU NVLink-connected system.**

***Index Terms*—PGAS, Deep Learning Recommendation Model, Collective Calls, Communication**

I. Introduction

Deep learning recommendation algorithms are widely used for a variety of products, including Google advertising, Netflix personalized video suggestions, and Instagram feeds. According to Meta's reports, more than 50% of machine learning training time at Meta is devoted to deep learning recommendation models (DLRM) [1], and over 70% of inference time is used for these models [2]. Managing and processing large-scale embedding tables during training and inference cycles is crucial, as they often become bottlenecks due to significant memory usage and intensive data transfer requirements.

In the domain of deep learning recommendation models (DLRM), embedding tables (EMBs) are a critical stage for capturing intricate relationships among entities such as users, items, and their respective sparse features. During the embedding table retrieval operations, these sparse feature inputs (categorical data) are converted into continuous representations, enabling neural networks to effectively discern complex patterns within the dataset. The memory usage of these embedding tables is typically vast, accounting for 99% of the memory used in DLRMs [3]. Driven by the need for better accuracy, the size of embedding tables continues to grow, often exceeding the limits of a single GPU's memory. Consequently, these embedding tables are frequently partitioned across multiple GPUs using model parallelism. This is also the major driving force to use multiple GPUs for DLRM.

Despite employing model parallelism for embedding tables, other layers of DLRM, such as the Multilayer Perceptron (MLP) layer, still use data parallelism to minimize communication and optimize performance. Using model parallelism for all stages would result in more communication overhead and yields lower overall performance. This mixture of two parallel computing strategies is the dominant approach for DLRM on multi-GPU systems and is the approach used in this paper. The use of these two strategies for parallelism necessitates converting the memory layout from model parallelism in the embedding table layer to one suitable for data parallelism. This conversion results in significant communication at the end of embedding table layers during both the inference forward pass and the training forward and backward pass, leading to a large fraction of overall runtime being spent on communication. Currently, the de facto communication scheme uses high-performance collective communication libraries like the NVIDIA Collective Communications Library (NCCL) [4]. As a result, both DLRM training and inference use a bulk synchronous model, calling collective communications at the end of the computation CUDA kernel, separating the program into compute and communicate phases. This execution model precludes the possibility of interleaving communication and computation at a fine granularity, yielding sub-optimal performance. Particularly, as the EMB layer of DLRM weak scales on more GPUs to meet the increasing memory size requirements of embedding tables, there is a significant gap between the achieved and perfect weak scaling behavior.

In this paper, we propose an alternative communication scheme to collective communication that utilizes Partitioned Global Address Space (PGAS) fine-grained one-sided messages. This scheme leverages GPU direct access instead of collective communication calls on the CPU side at the end of computation kernels (embedding table retrieval CUDA kernels). This approach offers three major benefits:

1) **Fine-Grained Communication and Computation Overlap:** In contrast to separate communication and computation phases, PGAS GPU direct communication sends many small one-sided messages immediately after the data becomes available inside the CUDA kernel, resulting in fine-grained communication and computation overlap.
2) **Smooth Network Usage:** Instead of sending aggregated messages in the communication phase, the PGAS direct GPU communication sends messages throughout the computation kernel as each piece of data is generated, thereby smoothing network usage.
3) **Eliminating Unpacking Step:** Collective calls move data in chunks, often requiring an additional step of unpacking the received data or rearranging it into local data structures (e.g., tensors). The PGAS direct GPU communication eliminates this local memory staging by writing data directly to remote device memory from thread registers without the need for unpacking.

We implement a PyTorch backend that uses the proposed PGAS direct GPU communication and test its effectiveness in a single-node, multi-GPU embedding retrieval forward pass on a 4-GPU NVLink-connected system. We compare it to a PyTorch baseline that uses NCCL for communication. In our experiments, our backend achieves a geometric mean speedup of 1.97x for the weak scaling test and 2.63x for the strong scaling test compared to the baseline.

We summarize our main contributions below:

1) We develop a PyTorch backend for implementing the embedding tables forward pass that supports PGAS direct-GPU communication.
2) Using the single-node, multi-GPU systems embedding retrieval application as a case study, we analyze the time components of the NCCL baseline, and demonstrate how several of the time components can be significantly reduced by switching to the PGAS direct-GPU communication scheme.
3) We experimentally verify the effectiveness of the PGAS direct-GPU communication scheme, achieving over 2x speedup over the NCCL baseline. Furthermore, we pinpoint that the performance benefits arise from intensive communication and computation overlap, as well as the removal of the unpack (data arrangement) step.

II. Background: DLRM Architectures

Figure 1 illustrates the high-level architecture of a DLRM [5]. DLRMs are used to make predictions based on domain-specific inputs, also referred to as samples. For example, Netflix uses these models for personalized entertainment recommendations, Airbnb for homestay suggestions, and Facebook for advertisement targeting. These inputs are generally categorized into **dense features** and **sparse features**, as shown in Figure 1. Dense features in a DLRM are continuous-valued attributes, represented as vectors of real numbers, capturing quantitative information. Examples include age, rating scores, and purchase amounts. Sparse features in a DLRM are categorical attributes represented as high-dimensional, sparse vectors with mostly zeros and few non-zero values corresponding to specific categories. These features encode qualitative information, such as pages browsed or users' IP addresses, and often require encoding techniques like one-hot encoding. The input for each sparse feature is often a bag of non-zero entries. The size of the bag, known as the pooling factor, varies by features and by samples.

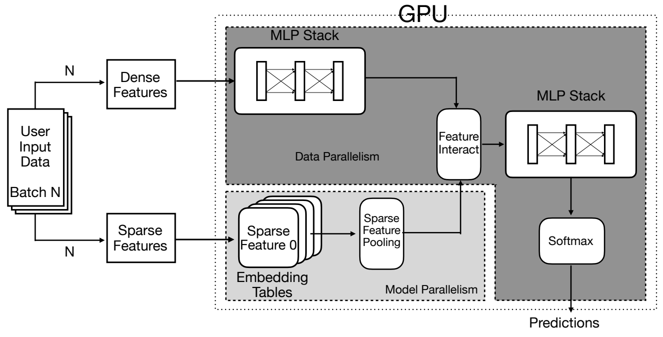

Fig. 1. Architecture of a DLRM. The dotted line outlines the stages of the DLRM that run on GPUs. The stages in the dark gray box are executed with data parallelism, while the stages in the light gray box are executed with model parallelism. The input data is generally partitioned on the CPU and then copied to the GPUs.

Dense feature inputs are fed into the top Multilayer Perceptron (MLP) layers, while sparse feature inputs are fed into the embedding (EMB) layer. Both the top MLP and the EMB layer generate batches of dense tensors, commonly referred to as **embeddings**. These embeddings are lower-dimensional, feature-rich representations that capture the essential characteristics of the input data. These embeddings pass through the interaction layer, which fuses the embeddings from the MLP and EMB layers using operations such as dot product, element-wise product, or concatenation to produce a single dense embedding. This single embedding is then fed to the bottom MLP. Finally, the output of the bottom MLP goes through a softmax layer to generate predictions in the form of a probability distribution over all possible outcomes. This workflow is shown in Figure 1.

A. *Embedding Tables (EMBs)*

At the core of DLRMs lie embedding tables, which store learned representations for sparse features. Figure 2 illustrates a typical embedding table structure, where each sparse feature corresponds to a distinct table. Each row within the table is an embedding vector, representing learned weights updated during backpropagation. The size of embedding vectors, denoted

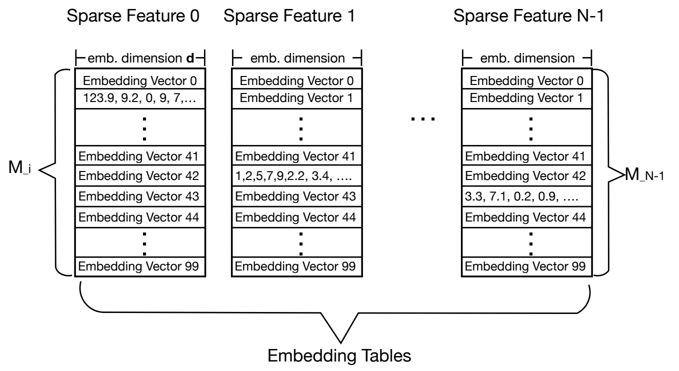

Fig. 2. Embedding tables for N sparse features. Each embedding table uses a hash size $M_i = 100$.

by d, is a hyperparameter that determines the dimensionality of the embedding space. Common values for d are powers of 2, such as 64, 128, or 256, for better memory alignment.

The size of each embedding table depends on the cardinality of the corresponding sparse feature space. Some tables, like those for US states, have small cardinalities (e.g., 50 rows). However, tables for features like user-browsed pages can have billions of rows, which is impractical due to memory constraints. To optimize memory, a hash function H : $\mathit{Cardinality}_{\text{sparse feature i}}$ maps sparse feature indices to $0, 1, \ldots, M$. This reduces memory demands. Each sparse feature may have different M values. Thus the total size of embedding tables scales approximately with $\sum_i d \times M_i$. The hash function also introduces hash collisions, where multiple sparse inputs map to the same embedding vector. However, collisions represent a necessary trade-off between prediction accuracy and memory usage. In recent years, the number of sparse features and the sizes of embedding tables have increased significantly to improve prediction accuracy. According to Facebook's records [6], the memory capacity requirements of DLRMs grew 16-fold between 2017 and 2021. Embedding tables now require terabytes of memory, accounting for roughly 99% of the overall model memory.

B. Embedding Table Lookup and Pooling Operations

Embedding table lookup and pooling operations are fundamental in the forward pass of the embedding table (EMB) retrieval. Figure 3 illustrates workflow of a typical EMB layer forward pass.

In this example, there are S sparse features, each corresponding to a embedding table in Figure 3. The input consists of three samples, forming a batch. We focus solely on the sparse input: each input sample may or may not have a sparse input for every sparse feature. As highlighted in a gray box in Figure 3, for the first sample, we have input for sparse features 1 and 2, but no input for sparse feature S. Each sparse feature with input typically consists of a bag of one or more indices. The size of this bag, often called the pooling factor, varies depending on the sparse feature and samples. In the example shown in Figure 3, sample 1 has 4 indices for sparse feature

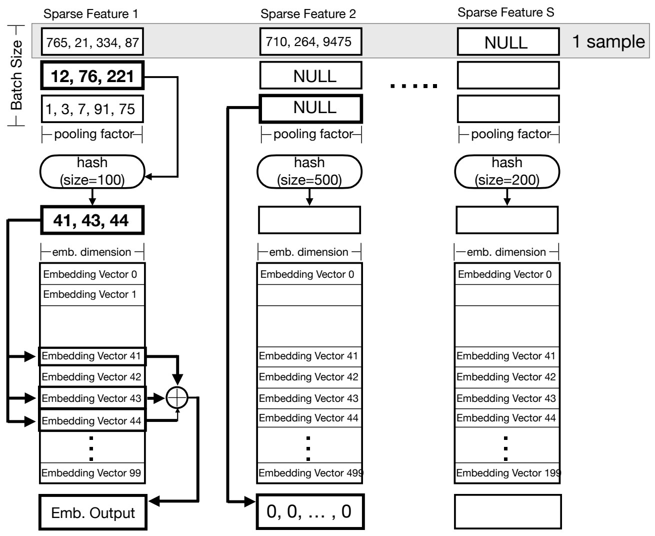

Fig. 3. Example illustrating the embedding lookup and pooling operation. The lookup and pooling operations are highlighted in bold. For sparse feature 1, the input is hashed, generating the index for the corresponding embedding table. Those embedding vectors are then read and combined via element-wise summation to produce the output vector of the lookup operation. In the case of sparse feature 2 and sample 3, the input data is NULL, indicating that sparse feature 2 contains no sparse input for sample 3.

1 and 3 indices for sparse feature 2. The following steps are then performed for each sparse input in the EMB layers:

1) **Hashing**: For each bag of indices for a particular sparse feature in a specific sample, apply the hash function H to all the indices in that bag, mapping them to the rows of the embedding table.
2) **Lookup**: Read the corresponding embedding vectors from the embedding table using the hash results as the indices.
3) **Pooling**[1]: Combine the embedding vectors obtained from a bag via element-wise summation to produce a single output embedding vector.

C. Distributed Forward Pass of DLRMs

As discussed in Section II-A, the entire set of embedding tables cannot fit into a single GPU memory, necessitating model parallelism to distribute embedding tables across multiple GPUs. Conversely, data parallelism is employed for other parts of the model, including the MLP layer and interaction layer, to optimize performance.

Data parallelism partitions the input by the batch dimension, while model parallelism partitions it by the embedding table dimension. Each GPU processes a full batch of specific sparse inputs whose embedding tables are located on that device, executing the EMB forward (or backward) pass in parallel.

In the example depicted in Figure 4, both sparse and dense inputs initially reside in the CPU memory with a batch size

[1]The terms "pooling operation" and "pooling factor" might be confusing. The pooling factor refers to the number that describes the bag size of a sparse input, while the pooling operation is the method used to combine multiple embedding vectors into a single embedding vector.

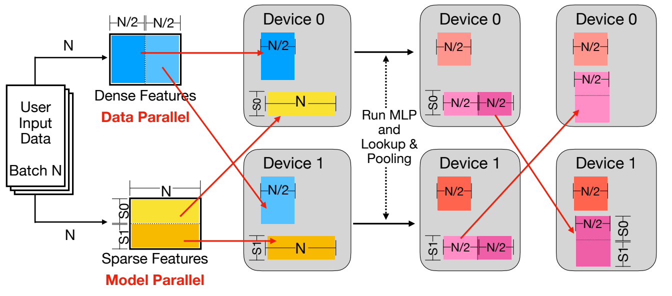

Fig. 4. Communication in the EMB layer forward pass. The dense inputs are partitioned into mini-batches. The sparse inputs are partitioned based on the location of each sparse feature: GPU 0 receives all sparse inputs for sparse feature 0, and GPU 1 receives all sparse inputs for sparse feature 1. The top MLP and EMB retrieval run concurrently. At the end of the EMB layer, GPU 0 sends a mini-batch of sparse feature 0 embeddings to GPU 1, and GPU 1 sends a mini-batch of sparse feature 1 embeddings to GPU 0. This allows the data layout to be suitable for data parallelism.

of N, and there are 2 GPUs in total. The dense inputs are divided into mini-batches, each with a mini-batch size of $N/2$. Conversely, the sparse inputs are partitioned based on the embedding table location. Assuming there are only 2 sparse features in this example, sparse feature 1 is located on GPU 0 and sparse feature 2 is located on GPU 1. The **full batch** of sparse inputs for sparse feature 1 is sent to GPU 0, and the **full batch** of sparse inputs for sparse feature 2 is sent to GPU 1. The dimensions of each partitioned dense input and sparse input are marked and shown in Figure 4.

At the end of the EMB layers, communication is required to convert the memory layout from model parallelism to a layout compatible with data parallelism. This involves repartitioning the EMB output embeddings along the batch size dimension and performing an all-to-all communication. Each GPU receives the output EMB embeddings of non-local sparse features that belong to the local mini-batch. As a result, each GPU has an $N/2$-sized mini-batch of embedding vectors for all sparse features. The communication required to achieve this data layout conversion, illustrated by the pink boxes in Figure 4, is the focus of our communication optimization.

III. Async One-sided Small Messages in EMB Forward Pass

A. Challenges in the EMB Forward Pass

In a typical PyTorch DLRM implementation, the transition from model parallelism to data parallelism is facilitated by collective all-to-all calls. Although local communication may overlap with remote computation, there is no overlap between local communication and computation. This clearly separated communication and computation phase scheme has the following potential performance drawbacks:

1) **Extra Unpacking Step**: An unpacking step (or data rearrangement) is often needed after collective communication calls, as continuous memory chunks are sent to maximize network bandwidth utilization.
2) **False Dependencies**: During the computation kernel, communication data is generated progressively. Aggregating this data for collective calls does not align well with the intrinsic data dependencies of the EMB layer forward pass. Each piece of communication data is ready to send immediately, without waiting for the generation of all other data. This mismatch introduces false dependencies and additional overhead in the communication control path, including the time spent waiting for all EMB layer output to be generated, CUDA kernel synchronization, and the time to trigger the collective calls.
3) **Dominant Synchronization Overhead**: With small batch sizes, the overhead of CUDA kernel synchronization can become significant compared to communication and computation, as the forward pass is essentially latency-limited.
4) **Lack of Fine-Grained Overlap**: The clear phase boundary between computation and communication eliminates the possibility of fine-grained communication and computation overlap.

B. PGAS One-sided Small Messages in EMB Forward Pass

Based on the previous discussion, the EMB layer can be optimized using fine-grained one-sided direct-GPU communication to fuse communication with computation. In this approach, each GPU thread can initiate PGAS small messages in the form of RDMA writes as soon as its necessary computations are finished, rather than waiting for all other messages to be generated. This PGAS approach has three primary advantages over collective communication calls:

1) **Elimination of Unpacking Step**: Messages are directly written to the remote GPU at the memory locations where they are supposed to be. This eliminates the need for the unpacking or data rearrangement step, along with the corresponding intermediate communication buffers.
2) **Reduced Communication Control Path Overhead**: Collective communication calls introduce extra overhead in their communication control path, including the time waiting for all other outputs to be generated and CUDA kernel synchronization. In the PGAS approach, messages are sent directly without waiting for any other operations. This eliminates the overhead of false dependencies from the communication control flow path. Consequently, the PGAS approach reduces the communication control path latency to the time of triggering the communication on the GPU. Although it is faster to trigger communication on the CPU than on the GPU, due to the various overheads incurred in collective communication calls, the overall communication control path latency of the PGAS approach is much lighter and faster than that of collective calls.
3) **Fine-Grained Overlap and Smoothed Out Network Usage**: The PGAS approach issues many small one-sided reads and writes. While collective calls make better use of network bandwidth by sending large chunks of messages, the PGAS approach sends messages immediately upon generation, spreading them out over the computation. This enables fine-grained communication

and computation overlap, avoids network congestion, and prevents the network bandwidth from becoming a bottleneck.

C. Implementation Details

We provide pseudocode for the customized PyTorch backend and the implementation of PGAS direct-GPU messaging in Listings 1 and 2. Listing 1 demonstrates a typical approach to invoking a customized CUDA/C++ backend. Within this listing, the C++ compiled backend function *torch.ops.PGAS.forward* is wrapped with *torch.autograd.Function* to handle both the forward and backward passes. *LookupFunction.apply* invokes the forward computation for the EMB layer in the DLRM forward function.

Listing 2 provides an implementation of PGAS direct-GPU messaging within the EMB forward CUDA kernel defined in *PGAS_EMB_forward_kernel*. When writing the result of an embedding lookup, the *GetEmbOwnerId* function queries the partition ID of the input. If the corresponding input belongs to the mini-batch of remote GPUs, CUDA threads write the resulting embeddings directly to the output array on the remote GPU with the correct index. This one-sided communication approach significantly reduces overhead (see item (2), Reduced Communciation Control Path Overhead, above).

Furthermore, the direct-GPU writes are issued by CUDA threads and leverage GPU instruction parallelism: they run concurrently and asynchronously with other CUDA threads, which enables significant computation-communication overlap (see item (3), Fine-Grained Overlap and Smoothed Out Network Usage, above).

Finally, the C++ function *PGAS_EMB_forward* invokes the CUDA kernel for each device, allowing concurrent execution, and then synchronizes all CUDA kernels to ensure the EMB layer forward pass is complete.

In summary, we expect this approach to allow for better utilization of available resources, reduce network idle time, eliminate unnecessary overhead, and enable more efficient overlap of communication and computation. Consequently, we can achieve improved performance and a reduction in the performance gap between achieved scalability and theoretical scalability, especially for weak scaling.

IV. Experimental Setup and Results

To evaluate the effectiveness of fine-grained one-sided asynchronous communication in the EMB layer forward pass on a multi-GPU system, we conduct a series of experiments.

Our baseline is a typical PyTorch implementation of the EMB layer forward pass, consisting of an EmbeddingBagCollection forward pass followed by the *all_to_all_single* collective call with *async_op* set to true. All input and output tensors are on GPUs, and with NCCL as the communication backend, these PyTorch calls invoke their corresponding CUDA and NCCL functions. On each GPU, communication does not start until the embedding table forward CUDA kernel finishes. We

Listing 1 PGAS direct-GPU-based EMB retrieval PyTorch code snippet.

```
class LookupFunction(torch.autograd.Function):
    @staticmethod
    def forward(
        ctx,
        weights,
        offsets,
        indices,
        outputs):
        # calling PGAS fused CUDA backend
        torch.ops.PGAS.forward(
            weight_lists,
            offsets,
            indices,
            outputs)

class DLRM(nn.Module):

    def forward(
        self,
        dense_input,
        sparse_input_offsets,
        sparse_intput_indices):
        # apply forward pass for top MLP layer with dense_input
        .....
        LookupFunction.apply(
            self.weights_lists,
            sparse_input_offsets,
            sparse_input_indices,
            outputs)
        # Continue next layer's forward with outputs
        .....
```

call *wait* after the *all_to_all_single* call to synchronize all the GPUs.

We modify the C++/CUDA backend of the baseline, implementing the approach described in Section III. This new backend eliminates the unpacking step and fuses communication and computation into a single CUDA kernel to maximize computation-communication overlap. We refer to this approach as the PGAS fused retrieval.

Both the baseline and PGAS fused implementations utilize the same codebase from the DLRM benchmark provided by Facebook and are run with identical configurations and identical dense and sparse inputs. For all experiments, the dense and sparse feature inputs are generated synthetically with a uniform random distribution. All experiments are conducted on a DGX machine with four V100 NVIDIA GPUs connected via NVLink. We run the full inference pipeline of the DLRM model [5] with 100 batches, measuring the accumulated time of embedding table forward pass and the subsequent communication and data unpacking and rearranging over the 100 batches. This focus allows us to better evaluate the effectiveness of the proposed asynchronous PGAS-style communication.

A. Weak Scaling Tests on up to 4 GPUs

Driven by the increasing memory requirements of embedding tables, we conduct weak scaling tests with both PGAS and baseline implementations on up to 4 GPUs. Each GPU handles 64 embedding tables, each with 1 million entries and an embedding vector size of 64. We use a batch size of 16,384

Listing 2 PGAS direct-GPU-based EMB retrieval CUDA backend code snippet.

```
template<typename scalar_t, typename AccuFunc, typename ShardFunc>
__global__ void PGAS_EMB_forward_kernel(
    const int n_gpus,
    const int gpu_id,
    const PackedTensorAccessor64<scalar_t, 2,
                                 RestrictPtrTraits> weights,
    const PackedTensorAccessor64<int64_t, 1,
                                 RestrictPtrTraits> offsets,
    const PackedTensorAccessor64<int64_t, 1,
                                 RestrictPtrTraits> indices,
    scalar_t * __restrict__ outputs,
    const int BatchSize,
    const int EMBDim,
    AccuFunc accu_func,
    ShardFunc GetEmbOwnerId) {
    // Compute the embedding lookup and pooling given sparse
    // input offsets and indices and store the embedding result
    // in Vec4T sum
    ......
    auto [pe, output_idx] = GetEmbOwnerId(
        threadIdx.y, blockIdx.x, BatchSize, n_gpus);
    if(pe == gpu_id) {
        // Writing the embeddings locally
        sum.store(outputs[output_idx]);
    }
    else {
        // PGAS writing the embeddings to remote GPU memory
        sum.store(outputs[output_idx], pe);
    }
}

void PGAS_EMB_forward(
    TensorList weights,
    TensorList offsets,
    TensorList indices,
    TensorList outputs) {
    int num_devices = weights.size();
    cudaStream_t streams[num_devices];
    for(int i=0; i<num_devices; i++) {
        // Compute threads and blocks size for kernel launch
        cudaSetDevice(i)
        cudaStreamCreateWithFlags(streams+i,
                                  cudaStreamNonBlocking);
        PGAS_EMB_forward_kernel<<<blocks, threads, 0,
                                  streams[i]>>>(....)
    }
    for (int i=0; i<num_devices; i++) {
        cudaSetDevice(i);
        cudaStreamSynchronize(streams[i]);
    }
}

TORCH_LIBRARY(PGAS, m) {
    m.def("forward", &PGAS_EMB_forward);
}
```

and a pooling factor generated from a uniform distribution with a maximum size of 128.

1) Performance Summary: The speedup of PGAS over the baseline is shown in the following table:

Speedup	2 GPUs	3 GPUS	4 GPUs
PGAS over baseline	2.10X	1.95X	1.87X

In general, our PGAS fused implementation achieves an average speedup of 1.97x over the baseline. Figure 5 shows the weak scaling factor for both PGAS and baseline with up to 4 GPUs. The weak scaling factor is calculated by dividing the runtime by single GPU runtime. The ideal weak scaling factor is a flat line with a value of 1.

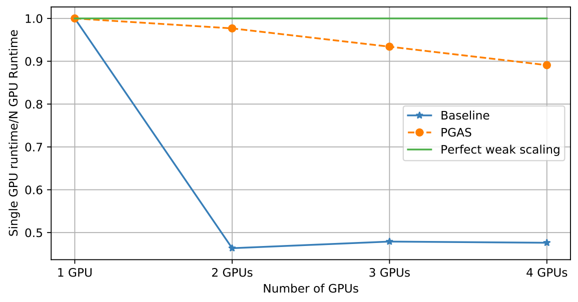

Fig. 5. Weak scaling tests for baseline and PGAS fused EMB retrieval.

Though the baseline's weak scaling flattens out beyond 2 GPUs, it has a significant gap compared to perfect weak scaling. In contrast, the PGAS implementation's weak scaling line is closer to the ideal.

The major weak scaling difference between PGAS fused and baseline implementations occurs when scaling from from 1 GPU to 2 GPUs. The significant jump in baseline is due to the additional communication time. Baseline uses bulk synchronization, which imposes a clear boundary between computation and communication phases. This inevitably introduces a communication time after the computation when we scale the system from single GPU to 2 GPUs. In contrast, the PGAS implementation can effectively hide the communication time within the existing computation. As we will see in Figure 6, the communication phase takes roughly the same time as the computation phase, so PGAS achieves almost perfect communication-computation overlap, incurring almost no additional cost when scaling from 1 GPU to 2 GPUs. Consequently, the baseline implementation has almost double the runtime of PGAS on 2 GPUs, resulting in a worse weak scaling factor of 0.46.

When going beyond 2 GPUs, Figure 5 shows that both PGAS and baseline have good scaling (shown by a relatively flat line). This is because communication time does not increase with more GPUs.

Now, we present a more detailed analysis of the runtime for PGAS and baseline.

2) Runtime Component Analysis: Figure 6 shows the PGAS runtime and breaks down the runtime for the baseline into three components: "Computation", "Communication", and "Sync + Unpack".

a) Method for measuring the time components for baseline: Measuring computation time is straightforward, as the baseline has distinct computation and communication phases. However, disentangling communication time from data unpacking time is difficult; this is because the baseline merges communication and data unpacking (or rearrangement) using an asynchronous request object. To overcome this, we first measure the entire communication phase with *wait* called at

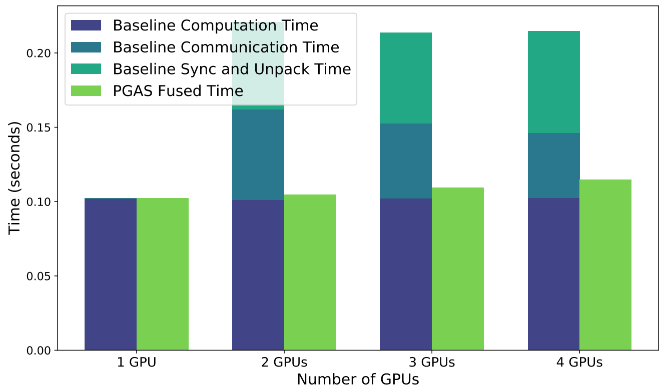

Fig. 6. Runtime and its time breakdown of PGAS and baseline implementation for the weak scaling test.

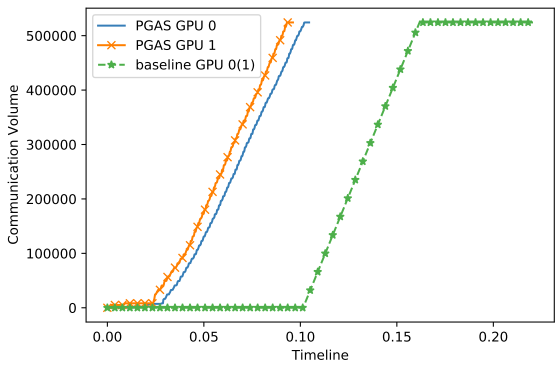

Fig. 7. Time profiling of communication volume over time for PGAS fused and baseline implementations on 2 GPUs.

the end. Then, we measure the same phase but only send a single float. By subtracting these two times, we estimate the synchronization and data rearrangement time; this is an overestimation by only the latency of sending a single float. The communication time is then calculated by subtracting the synchronization and data rearrangement time from the total communication phase time, resulting in an underestimated value since network latency is excluded.

b) Method of measuring communication for PGAS fused implementation: It is challenging to provide a similar breakdown for the PGAS implementation due to the heavy interleaving of communication and computation. Instead, we show the communication volume over time for the PGAS implementation in Figure 7. The x-axis represents the timeline, and the y-axis represents communication volume in units of 256 bytes. We designed a communication counter to be read every hundred GPU clock cycles. With each RDMA write, that thread also atomically adds to that counter. Consequently, sequential reads of the communication counter show the communication volume over time. For comparison, we also plotted the communication volume over time for the baseline implementation, represented by the dashed green line. The communication volume is calculated by linearly interpolating the total communication volume over the communication time.

c) Baseline: The baseline runtime comprises three major parts: (1) **computation**, (2) **communication**, and (3) **data unpack time and CUDA synchronization overhead**. Weak scaling tests maintain a constant workload per device while increasing the problem size with the number of GPUs. Consequently, as the number of GPUs increase during weak scaling tests, we expect:

- **Computation time stays the same,** because computation workload per-GPU stays the same.
- **Communication time decreases.** Although total communication volume increases, as GPUs are added, $\frac{\text{total communication volume}}{\text{total network bandwidth}}$ decreases. Thus the communication time should decrease with more GPUs.
- **Sync and unpack time increases** because there is more received communication data, leading to increased unpack workload.

These expectations are consistent with Figure 6. The increase in the sync and unpack time roughly balances out the decrease in communication time, thus the total runtime stays roughly the same. This in turn leads to good weak scalability beyond 2 GPUs.

d) PGAS Fused Implementation: The PGAS fused implementation is about twice as fast as the baseline. From Figure 6, PGAS fused time is only slightly more than the baseline computation time. The PGAS fused implementation writes the data directly to remote GPU memory inside the CUDA kernel. Those writes are small messages that are generated by the intricate computation pattern and are sent away immediately without any explicit aggregation. Note GPU memory warp coalescing (handled by hardware) is still in effect, aggregating the message with natural locality. As a result, the PGAS fused implementation is able to significantly overlap communication with computation. From the corresponding baseline time components, the communication time is less than the its computation time. PGAS fused implementation is able to almost completely hide the communication. More straightforward evidence is seen in Figure 7: the communication volume is well-distributed over the computation time and largely overlaps with computation. In contrast, the baseline implementation has a long initial period when communication volume stays flat at 0.

Additionally, the PGAS fused implementation writes the data directly to the location needed for the next DLRM layer, eliminating the need for data unpacking and rearrangement. Those remote writes may lead to some random access, but they run concurrently with local computation and are therefore hidden and do not increase runtime.

Now, let's go back to the weak scaling result (Figure 5) to answer the question "why does the PGAS fused implementation not have the weak scaling performance decline between 1 and 2 GPUs, as baseline does?" This is because PGAS

1) **Removes** the need for unpack (data rearrangement), and
2) **Overlaps** the communication with the computation.

We observe that the runtime of the PGAS fused implementation slightly increases with more GPUs used. The increased

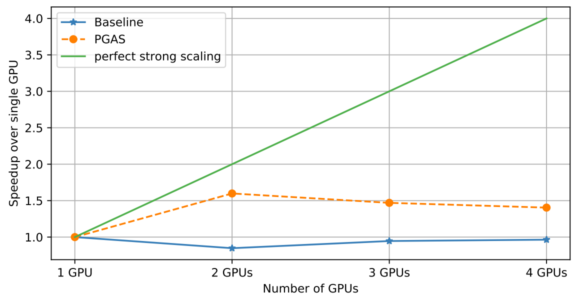

Fig. 8. Strong scaling test on up to 4 GPUs connected via NVLink.

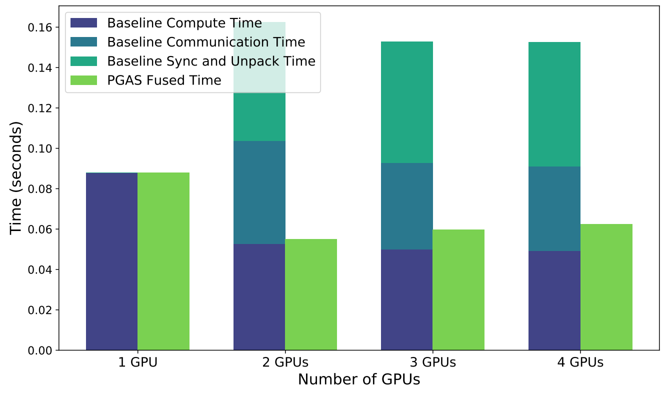

Fig. 9. Runtime and time breakdown of PGAS and baseline implementation for the strong scaling test.

time comes from the increase communication overhead. PGAS fused implementation uses many small messages. Compared to large messages, those small messages are not bandwidth-efficient as the message header takes a good portion of bandwidth. This header overhead increases as the communication volume (total number of small messages) increases. However, the overhead only increases very slightly with more GPUs. The reason is as follows: By intensively overlapping communication with computation, the PGAS fused implementation is not bandwidth-limited as long as the communication can be done within the computation period. Thus the bandwidth inefficiency does not readily translate to an increase in the overall runtime. Although small messages are not bandwidth-efficient, the use of small messages is able to smooth out the network usage.

B. Strong Scaling Tests on up to 4 GPUs

We conduct strong scaling experiments with both the baseline and the PGAS fused implementation on up to 4 GPUs. In this strong scaling test, the total workload is limited by the single GPU memory (32 GB). Therefore, we choose the configuration with 96 embedding tables, each with 1 million entries and an embedding vector size of 64, a batch size of 16,384, and a pooling factor of up to 32 to maximize GPU memory usage.

1) Performance Summary: The speedup of PGAS over baseline is shown in the following table:

Speedup	2 GPUs	3 GPUS	4 GPUs
PGAS over baseline	2.95X	2.55X	2.44X

On average, the PGAS fused implementation achieves a speedup of 2.63X over baseline. Figure 8 shows the strong scaling factors for both PGAS and baseline with up to 4 GPUs. The strong scaling factor is calculated by dividing the single GPU runtime by the runtime, namely the speedup over single GPU implementation. The ideal strong scaling line is marked out in green.

Neither PGAS nor baseline achieve good strong scaling: baseline with {2,3,4} GPUs were all slower than baseline on single GPU. PGAS has slightly better strong scaling, with {2,3,4} GPUs all faster than a single GPU. However, the strong scaling for PGAS decreases beyond 2 GPUs.

The reason is that embedding table retrieval features low arithmetic intensity, the computation workload is mainly memory operations and is proportional to the batch size and the number of local embedding tables. Strong scaling tests maintain the **total workload constant** and evaluate the speedup as we partition the problem and run it on more GPUs. In our strong scaling test, the total workload is limited by the single GPU memory size. As a result, as we partition the workload on more GPUs, the computation workload per GPU is not enough to fully utilize either the GPU computation resource or the GPU memory bandwidth; thus adding more GPUs does not lead to acceleration. This is verified by the computation throughput and memory throughput collected on the computation kernel on 2 GPUs with the NVIDIA profiling tool *ncu*. Both computation throughput and memory throughput are less than 60% with 38% for computation throughput and 57% for memory throughput.

Despite the difficulty in accelerating the EMB forward pass with more GPUs, PGAS still achieves better strong scaling than baseline, especially from single GPU to 2 GPUs. This is because in **baseline**, the decrease in computation time is balanced out by the increase in communication time, synchronization, and unpacking time. As a result, the total runtime increases by 1.8x, failing to achieve any speedup with multiple GPUs. On the other hand, the PGAS fused implementation is able to hide the increased communication with computation, achieving a roughly 1.6x speedup over a single GPU.

2) Runtime Component Analysis: Figure 9 shows the PGAS runtime and the time breaks down for the baseline.

a) Baseline: Strong scaling tests maintain a constant **total** workload while partitioning the fixed workload on more GPUs. Consequently, as the number of GPUs increases during strong scaling tests, we expect:

- **Computation time decreases with 2 GPUs and stays roughly the same beyond 3 GPUs** because the computation kernel used in this experiment is latency-limited beyond 2 GPUs.
- **Communication time decreases**. Although total communication volume increases, as GPUs are added,

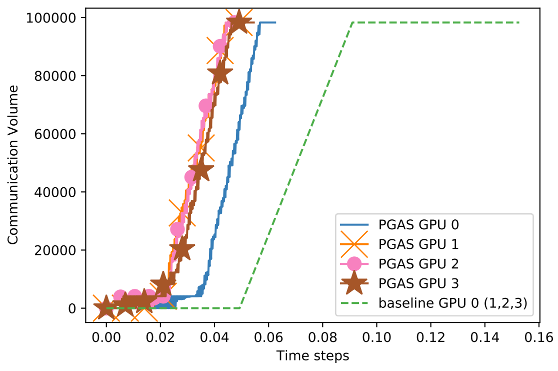

Fig. 10. Time profiling of communication volume over time for PGAS and baseline implementations on 4 GPUs.

$\frac{\text{total communication volume}}{\text{total network bandwidth}}$ decreases. Thus the communication time should decrease with more GPUs.

- **Sync and unpack time increases** because there is more received communication data, leading to increased unpack workload.

These expectations are consistent with Figure 9. While the computation time decreases with 2 GPUs, the incurred sync and unpack time, along with the communication time, together offset the decreased computation time, resulting in a higher runtime than that of a single GPU. Beyond 2 GPUs, the increase in the sync and unpack time roughly balances out the decrease in communication time with the unchanged computation time beyond 2 GPUs, resulting in a total runtime that stays roughly the same. Both of those in turn lead to no speedup or sometimes, a slowdown under strong scalability.

b) PGAS Fused Implementation: The PGAS fused implementation is on average 2.63x faster than the baseline. From Figure 9, the *total time* for PGAS fused is only slightly more than the *computation time* for baseline. This is once again because the PGAS fused implementation is able to almost completely hide the communication with computation. (From the time breakdown for baseline in Figure 9, we infer that communication time is less than computation time.) More straightforward evidence is seen in Figure 10: the communication volume is well-distributed over the computation time and largely overlaps with computation on 4 GPUs. Finally, the PGAS fused implementation writes directly to the final remote memory location, removing the need to unpack and rearrange data that is incurred in the baseline.

We observe that the runtime of the PGAS fused implementation slightly increases with more GPUs. The reason is the same as in the weak scaling test; since small messages are not bandwidth-efficient, the increased total communication volume introduces more overhead. However, just as with weak scaling, those increased overheads do not directly translate to increased runtime but instead can be hidden by computation.

In general, by removing the unpack (data rearrangement) step and overlapping the communication with computation intensively, the PGAS fused implementation achieves on average 2.63x speedup over the baseline and 1.6x strong scaling speedup on 2 GPUs. However, because the computation kernel becomes latency-limited, the strong scaling speedup does not continue with more GPUs.

V. Conclusion and Future Work

In this paper, we demonstrate the effectiveness of adopting PGAS-style one-sided small direct-GPU messages in the forward pass of multi-GPU embedding table retrieval. We analyze the time components of the traditional programming model, which involves a CUDA kernel backend followed by a non-blocking collective call, and verify that PGAS-style direct-GPU messages can address performance issues in this approach. Specifically, PGAS-style one-sided small messages can hide communication within computation, significantly reducing communication time. By writing directly to the final remote memory location, they eliminate the unpacking (data rearrangement) step required in the traditional approach. Additionally, these messages smooth out network usage by distributing communication across the entire computation period. As a result, we observe runtime speedups and better weak scalability in the backend using the PGAS-style direct-GPU message scheme compared to the traditional approach.

In the future, we intend to apply the PGAS-style direct-GPU messaging scheme to the EMB layer's backward pass, which also faces significant scalability issues and performance bottlenecks due to communication and synchronization. During the forward pass, each GPU receives the necessary embeddings for its portion of the input data, with communication volume proportional to the embedding dimension size and batch size. However, during backpropagation, the gradients for these embeddings need to be communicated back to the GPUs that own these embeddings. The communication volume in the backward pass is proportional to the embedding dimension size and the number of unique embeddings needed for the batch, which can be as large as the embedding table size in the worst case. This volume is typically much higher than in the forward pass.

Additionally, during the backward pass, gradients must be aggregated across all GPUs that used the same embeddings. If multiple GPUs used the same embedding, their gradients must be summed to get the total gradient for that embedding. Combining these gradients often requires multiple rounds of collective calls, where embeddings are shifted to (received from) the next (previous) GPU, a step not needed in the forward pass. This process necessitates multiple synchronizations to ensure all GPUs have consistent gradient information before shifting and finally updating the embeddings, adding significant synchronization overhead.

We believe that the PGAS-style direct-GPU messaging scheme can optimize communication by replacing multiple rounds of collective calls with atomic PGAS direct-GPU remote writes. By achieving better overlap of communication and computation (since gradient computation is more complex and thus takes more time, leaving a larger window to hide communication), and by replacing multiple rounds of synchronization with atomic operations, we can substantially reduce communication and synchronization time. This approach is expected to lead to a significant performance improvement in the EMB layer backward pass.

In our current implementation, we partition the sparse inputs on the CPU and then copy it to the GPU. The time spent on input partitioning is small in our experiments because we use a simple table sharding scheme (partitioning by tables). However, if a more complicated sharding scheme is used (partitioning by rows) [6], the sparse input partitioning and aggregation time will become more significant. A potential optimization is to merge the sparse input partitioning into the computation kernel, allowing computation to start immediately when the corresponding sparse input is picked out. This optimization can achieve multiple overlaps: sparse input processing with computation, sparse input processing with communication, and communication with computation, thus improving overall performance.

The current implementation is designed for a single-node, multi-GPU system. We also aim to extend this work to a multi-node system. Compared to high-performance NVLinks, inter-node connections have higher latency and lower bandwidth, which could lead to reduced bandwidth utilization and impact overall performance in a PGAS-style communication setup. To mitigate this, an asynchronous communication aggregator [7] can be employed to improve bandwidth utilization with minimal changes to the existing code. The modification involves replacing the operation *sum.store(outputs[output_idx], pe)* with *aggregator.store(outputs[output_idx], sum, pe)*, where data is written to the aggregation buffer instead of directly to remote GPU memory. The aggregator then transmits the aggregated message to remote memory based on user-defined aggregation size and maximum wait time.

Acknowledgements

This work is supported by the National Science Foundation (NSF) under projects CCF-1823034 and CCF-1823037 and in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

We would like to thank Louis Feng from Facebook for his guidance on this work, and Collin McCarthy from Owens' group for his consistent support.

References

[1] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Feb. 2021, pp. 802–814.

[2] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "DeepRecSys: A system for optimizing end-to-end at-scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, May 2020, pp. 982–995.

[3] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of Facebook's DNN-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2020, pp. 488–501.

[4] NVIDIA, "NVIDIA collective communications library (NCCL)," https://developer.nvidia.com/nccl, 2024, [Accessed: July 20, 2024].

[5] Meta, "An implementation of a deep learning recommendation model (DLRM)," https://github.com/facebookresearch/dlrm, 2024, [Accessed: July 20, 2024].

[6] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C.-J. Wu, "RecShard: statistical feature-based memory optimization for industry-scale neural recommendation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. ACM, Feb. 2022, pp. 344–358.

[7] Y. Chen, B. Brock, S. Porumbescu, A. Buluç, K. Yelick, and J. D. Owens, "Scalable irregular parallelism with GPUs: Getting CPUs out of the way," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22, Nov. 2022, pp. 708–723.