Domino: Eliminating Communication in LLM Training via Generic Tensor Slicing and Overlapping

Guanhua Wang, Chengming Zhang, Zheyu Shen*, Ang Li*, Olatunji Ruwase

Microsoft DeepSpeed

Abstract

Given the popularity of generative AI, Large Language Models (LLMs) often consume hundreds or thousands of GPUs for parallelizing and accelerating the training process. Communication overhead becomes more pronounced when training LLMs at scale. To eliminate communication overhead in distributed LLM training, we propose Domino, which provides a generic scheme to hide communication behind computation. By breaking data dependency of a single batch training into smaller independent pieces, Domino pipelines these independent pieces training and provides generic strategy of fine-grained communication and computation overlapping. Extensive results show that, comparing with Megatron-LM, Domino achieves up to 1.3x speedup for LLM training on Nvidia DGX-H100 GPUs.

1 Introduction

Recent advances in Generative AI (GenAI) enable new application scenarios in various domains, such as chat-bot [53], text generation and summary [11, 52], image and video content creation [62]. These GenAI applications are based on carefully trained foundation models as large language models (LLMs). Well-established LLMs are transformer models, such as GPT [11, 52, 60] and Llama [5, 6, 72, 73] series. Given LLM model sizes are usually ranging from tens to hundreds of billion parameters which is far exceeding a single GPU's memory and computation limit, distributed model training over hundreds to thousands of GPUs is necessary.

For LLM transformer training, three prominent paradigms have emerged: data parallelism (DP), tensor parallelism (TP) and pipeline parallelism (PP). Vanilla data parallel training refers to every GPU maintaining a full copy of whole model parameters but training on different input data. Model parameter need to be synchronized at the end of each training iteration among all GPUs in use. To mitigate memory pressure from LLMs' huge volume of parameters in DP training, ZeRO [61] or FSDP [81] is often needed to spread model parameters among all GPUs and only recollect the parameters needed to conduct computation. TP and PP belong to model parallelism, where PP [24, 26] partitions different model layers on different GPUs. Instead of different GPU holding different model layers, TP [33, 67] splits every layer on each GPU, and thus every GPU holds one portion of every model layer.

*University of Maryland

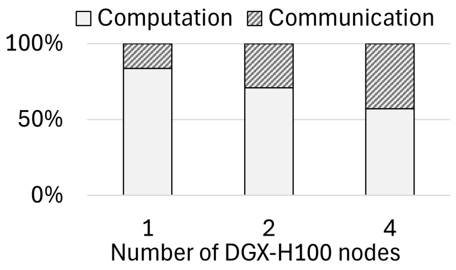


Figure 1: GPT-3-13B computation and communication ratio per training iteration over 1 DGX-H100 node (8 H100), 2 nodes (16 H100) and 4 nodes (32 H100) using TP.

Among all distributed model training paradigms, tensor parallelism (TP) has garnered increasing popularity, especially on Nvidia GPUs [21, 22]. TP was standard practice for single-node multi-GPU training, given its decent system efficiency in high communication bandwidth (i.e., NVlink [50], NVSwitch [51]) cases. However, because of limited cross-node network bandwidth, TP falls short in multi-node cases. Recently, Nvidia is breaking the bandwidth gap between inter-node and intra-node links. For example, the latest DGX-H100 [22] box is equipped with high-bandwidth Infiniband (IB) links with an aggregated bandwidth of 400 GB/s for cross-node communication, which is at the same level of intra-node NVSwitch bandwidth (i.e., 900GB/s on DGX-H100 [22]). Therefore, it is time to optimize and propose a TP-only solution for LLM training that covers both single-node and multi-node scenarios.

The major overhead of TP is its per-layer global communication, which lies on the critical path of training execution.

As described in the literature [67], every transformer layer needs to communicate twice in forward and another twice in the backward using NCCL collectives [30] (§ 2.3). Given these collective communications happen on the critical path of execution, it is hard to hide these communications behind successive computations with the general communication overlapping strategy used in DP [61] or PP [24] training process. Prior arts [54, 55] report this communication overhead can be up to 45% of end-to-end training iteration time. As one of our measurements depicted in Figure 1, even with the lastest DGX-H100 nodes connected with 400GB/s IB, communication still takes from 17% to 43% of end-to-end GPT-3-13B training iteration time. Furthermore, the communication ratio would continue to grow when scale up to more nodes. To mitigate this high communication overhead in TP, prior work [54, 78] provides kernel fusion of a GeMM (General Matrix Multiplication) with its subsequent collective calls (e.g., NCCL [30]) to achieve fine-grained computation-communication overlapping. However, this type of kernel fusion technique limits the overlapping scope and is not general enough to always hide communication behind computation. Especially in the cases where collective communication takes much longer than a single GeMM computation, most of the communication time still stands out as the major training overhead. Furthermore, given that computation on the latest GPUs (e.g., DGX-H100 [22], DGX-B200 [45]) is becoming faster, communication overhead is more pronounced in both single node and multi-node cases.

To provide a generic approach of hiding communication behind computation in TP, we propose Domino, a generic approach that breaks data dependency of transformer model training into pieces, and then pipelines these pieces training to overlap communication with computation. Besides traditionally TP can only be used within a node, Domino provides a uniformed TP solution for both single-node multi-GPU and multi-node multi-GPU cases. Compared with previous GeMM+NCCL fusion solutions, Domino provides a much wider scope of computation and communication overlapping (e.g., AllReduce not only overlaps with a single GeMM, but also LayerNorm, DropOut, etc). Additionally, any kernel fusion and optimization techniques can be easily integrated with Domino as a drop-in replacement to further boost overall system efficiency.

Extensive benchmark results are collected from Nvidia latest hardware DGX-H100 boxes, which are connected with 3200 Gb/s (i.e., 400 GB/s) InfiniBand (IB) fabrics [46]. We benchmark training with popular transformer models such as GPT-3 [11] and Llama-2 [73]. Compared with state-of-the-art TP implementation Megatron-LM from Nvidia, Domino achieves up to 1.3x speedup for both single-node and multi-node cases. Overall, Domino provides a generic approach of flexible overlapping of communication with a wide range of computation kernels for transformer training.

We summarize our key contributions as follows:

- To the best of our knowledge, Domino is the first work providing an end-to-end solution of generic communication-computation overlapping for tensor-parallelism-only training in both single-node and multi-node cases.

- Compared with previous arts, Domino provides a more flexible and wider range of computation and communication overlapping strategies.

- Experiment results on DGX-H100 boxes show that, compared with Megatron-LM, Domino achieves up to 1.3x speedup for GPT and Llama models training.

- Domino will be open-sourced and released as part of `https://github.com/microsoft/DeepSpeed`

2 Background and Motivation

In this section, we first describe the most widely used distributed transformer training schemes as data parallelism (DP), tensor parallelism (TP) and pipeline parallelism (PP) as § 2.1. Then we illustrate why TP is becoming increasingly popular among these three approaches as § 2.2. Finally, we analyze the communication overhead of TP in both single-node multi-GPU and multi-node multi-GPU cases in § 2.3.

2.1 Distributed Training Schemes

There are mainly three different kinds of paradigms for distributed LLM training, namely, data parallelism (DP), tensor parallelism (TP), and pipeline parallelism (PP).

In vanilla DP, each GPU maintains a full copy of model weights and consumes different input data. At the end of each training iteration, all GPUs involved conduct an AllReduce operation to synchronize model parameters. Specifically for transformer models, ZeRO [61] and FSDP [81] are widely used to reduce memory pressure on devices. In these fully shared data parallel schemes, whole model weights are often evenly split across all GPUs. When computation needs to happen on a specific layer, every GPU recollects the full weights of this particular layer by conducting an AllGather operation among all GPUs for this layer. Once the computation is done, every GPU releases the whole layer weights and only maintains the portion of weights that were originally assigned on each GPU. Therefore, ZeRO/FSDP can be regarded as a memory efficient data parallelism scheme that trades more communication with less on-device memory footprint.

PP and TP are both representative of model parallelism techniques. PP partitions a layer or a group of layers on a single GPU and then pipeline executes from GPU holding the first layer to GPU holding the last layer during forward propagation, and then backward propagation in the reverse order. Compared with PP, TP partitions the model in an orthogonal direction, where each GPU holds a portion of every

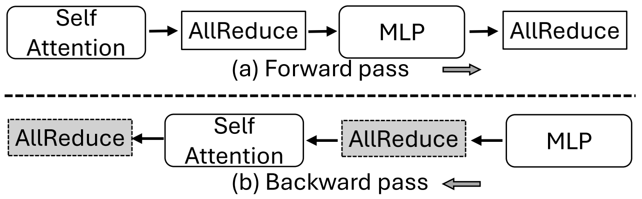


Figure 2: 4 AllReduce in each transformer block in TP training. Two blank AllReduce boxes are in forward pass, and the other two grey AllReduce boxes are in backward pass.

model layer, such that every GPU can compute from the first layer to the last layer by itself without blocking caused by sequential dependency created among PP stages [36]. Additionally, TP seems to have a similar model partition strategy as ZeRO/FSDP. The main difference here is, compared with ZeRO/FSDP, TP never recollects weights during forward or backward computation but synchronizes on activations or gradients via AllReduce. Compared with DP and PP, TP provides the highest system efficiency or training throughput with high bandwidth communication links [2, 70, 74].

2.2 TP is Trending

Tensor parallelism is gaining popularity for LLM workloads on Nvidia GPUs. AI practitioners have recently witnessed substantial improvements in both TP software and hardware stacks.

On the software side, Nvidia consistently enhances its Megatron-LM [33, 67] software stack as the state-of-the-art TP implementation. Megatron-LM achieves greater efficiency through integrating with more fine-tuned and customized compute kernels sourced from libraries like apex [44], cutlass [17], cublas [15], cudnn [16]. Besides that, Megatron-LM also involves new features to enhance overall system throughput, including selective activation checkpointing, and sequence parallelism strategies [33].

More importantly on the hardware front, Nvidia is pushing hard to bridge the bandwidth gap between intra-node and cross-node links, which is essential for extending TP to cross-node use cases. The latest DGX-H100 node is equipped with eight Nvidia ConnectX-7 InfiniBand (IB) cards [42], and each provides 400 Gb/s bandwidth. Thus each DGX-H100 box achieves 400 GB/s cross-node communication bandwidth, which is comparable to intra-node NVLink/NVSwitch bandwidth as 900 GB/s [22]. Furthermore, advancements in Nvidia's network infrastructure suggest that future DGX systems could potentially integrated with Nvidia ConnectX-8 IB cards [48], offering up to 800 GB/s aggregated cross-node bandwidth, approaching the bandwidth available with intra-node NVLink/NVSwitch connections.

With these advancements in software and hardware, both PyTorch [69] and raising vLLM [75] communities lean towards applying TP for both transformer training and inference. For instance, the PyTorch team sets TP as one major future direction of efficient LLM training in their recent release [2], and also sets improving TP scalibility as Key Results (KR) in Meta PyTorch team 2024 H2 roadmaps [70]. Similarly, on the inference side, vLLM has embraced TP as the only option for distributed LLMs serving [74].

Given the growing popularity of TP and recent breakthroughs in IB hardware, it is now imperative to establish a uniformed LLM training solution of TP for both single-node and cross-node scenarios. Before delving into our Domino design, we next discuss the communication overhead associated with TP.

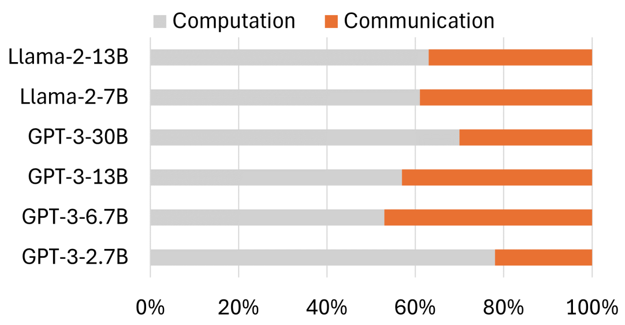


Figure 3: TP computation and communication ratio per training iteration on varied model types and model sizes over 1 to 4 DGX-H100 nodes (8 to 32 H100 GPUs).

2.3 TP Communication Overhead

We conduct measurements using state-of-the-art TP implementation from Nvidia as Megatron-LM [33, 47, 67].

TP communication possesses more unique characteristics compared with PP or DP solutions, mainly because *it resides on the critical path of every input batch training*. Hiding communication behind computation is standard practice not only limited to LLM training but also extensively applied in all distributed system environments [18, 23, 38, 77, 82]. For transformer training using DP or PP, the overlapping of communication and computation is quite straightforward since we can schedule communication on a side channel thus bypassing the critical execution path. For DP approaches like ZeRO [61] or FSDP [81], pre-fetching weights enable overlap with computation, as weights inherently do not have any sequential data dependency. PP naturally overlaps communication and computation by processing on different input batches. For instance, on each GPU, PP can overlap the previous batch's communication with computation for current batch data.

As described in Megatron-LM [47, 67], each transformer block comprises a self-attention layer and an MLP (multilayer perceptron) layer. As shown in Figure 2, both self-attention and MLP layers trigger an AllReudce operation

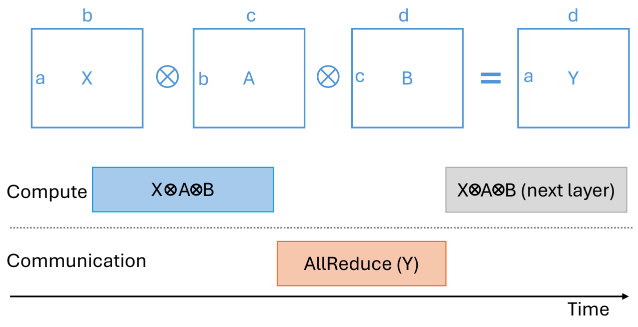


Figure 4: Forward pass of single Self-Attention / MLP layer.

in both forward and backward propagation. Consequently, each transformer block necessitates a total of 4 AllReduce per each training iteration. Given a language model consisting of N stacked transformer blocks, this results in $4 \times N$ AllReduce per iteration, imposing significant communication overhead. Furthermore, as discussed above, traditional methods fail to hide this communication behind computation, thereby placing all TP communication overheads on the execution critical path.

We measure the communication overhead in TP training with Megatron-LM across GPT-3 and Llama-2 model series with different model sizes. The models run on different numbers of DGX-H100 nodes ranging from 1 to 4 (i.e., 8 to 32 H100 GPUs) depending on model sizes and batch sizes. As shown in Figure 3, the communication time is ranging from 22% to 47% of end-to-end iteration time. This finding underscores that, despite the utilization of high bandwidth NVLink/NVSwitch/Infiniband interconnects, the communication overhead remains a significant portion of training iteration time. This is primarily due to more significant increase in computation power per each GPU (e.g., H100) compared with previous generations (e.g., V100, A100), thereby making communication overhead still stand out.

3 Domino Design

In this section, we describe the detailed design of Domino architecture. We first provide an overview of system architecture (§ 3.1). Then we detail how to generically partition computation and overlap sequences of computation kernels with communication (§ 3.2,§ 3.3, § 3.4).

3.1 Overview

We first describe overall workflow of Domino. Given standard transformer architecture, we abstract both self-attention layer and MLP (multi-layer perception) layer as weight tensors of A_FULL and B_FULL, where A_FULL stands for attention weights (i.e., W_q,W_k,W_v) for self-attention layer but linear weights for MLP, and B_FULL is linear weights for both self-attention and MLP layer. For ease of illustration, we describe our partition strategy in forward propagation since backward propagation is just in reverse execution order. Given layer input data X, both self-attention and MLP layers' computation can be abstracted as Equation 1.

$$X \otimes A\_FULL \otimes B\_FULL = Y\_FULL \tag{1}$$

As shown in Figure 4, TP (e.g., Megatron-LM) splits whole weights tensor A_FULL *column-wise* as set$\{A \mid A_i \text{ on } GPU_i\}$, and weights tensor B_FULL *row-wise* as set $\{B \mid B_i \text{ on } GPU_i\}$ for both self-attention and MLP layers. After every GPU getting its own A and B weights partitions, TP executes $X \otimes A \otimes B = Y$ and then conducts AllReduce on set $\{Y \mid Y_i \text{ on } GPU_i\}$ sequentially to recover Y_FULL, which makes communication overhead completely stand-out.

To hide TP communication behind computation, Domino provides extra and generic tensor partition in two dimensions on every GPU: *row-wise* split on inputs X and *column-wise* split on weights B on top of original TP model partitions.

At high level, Domino generically breaks TP's $X \otimes A \otimes B$ into smaller compute units without data dependency. Then we pipeline these independent compute units with collective communication to achieve fine-grained computation and communication overlapping. With the latest trend that attention computation is modularized and highly optimized like flash-attention [19, 20], windowed-attention [10], etc., we keep A untouched and do not conduct any tensor partitioning on A. Therefore, we only conduct tensor slicing on input tensor X (§ 3.2) and the second group of linear weights as B (§ 3.3). We also provide a hybrid tensor partition strategy of both X and B (§ 3.4). After these tensor slicing, Domino breaks $X \otimes A \otimes B$ into pieces and removes data dependency. Then we enable computation-communication overlapping on these independent pieces to reduce communication overhead in TP.

Prior to real model training, we benchmark system efficiency to determine the tensor partition granularity with grid search. These benchmarks guide our selection of tensor computation sizes to ensure minimal impact on computation kernel efficiency. To further enhance system efficiency, Domino also adopts the latest features like kernel fusion, torch.compile [59] and CUDAGraph [43, 58] techniques from PyTorch and Nvidia as described in § 4.3.

3.2 Row Split on Inputs

We first discuss row-wise split on input data, which refers to tensor partitioning on X in § 3.1. For ease of illustration in Figure 5, we simplify and assume all tensors are with 2 dimensions. Then X can be split in either row dimension or column dimension. In reality, each layer's input tensor usually is 3D as $(batch, seq, hidden)$. We map row/column dimensions of our example 2D tensor into batch/hidden dimensions of real 3D tensor, respectively.

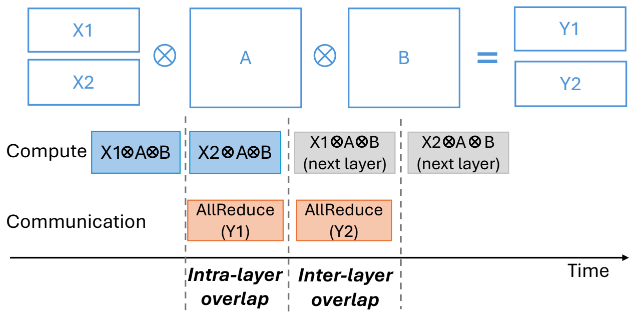


Figure 5: Domino row-wise (batch-dim) split on input X.

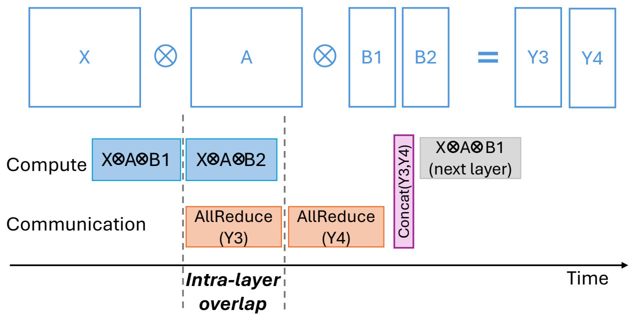


Figure 6: Domino col-wise (last-dim) split on weights B.

If we split input in column dimension to N chunks, the communication volume will be N^2 times bigger than vanilla baseline. As shown in Figure 4, assuming X with tensor shape as (a,b), A as (b,c) and B as (c,d). If we do a column-wise split on X and shard it as $[X_1, X_2 ... X_N]$ with each shape of $(a, b/N)$, we will get N^2 output tensors with original Y shape (a,d) after $X \otimes A \otimes B = Y$ with proper reshaping on A. To avoid this communication volume blow-up, we choose row-wise split X to $(a/N, b)$ in Figure 5, which refers to input tensor partition $(X1, X2)$ on batch dimension in reality.

$$X \otimes A = \begin{cases} softmax(\frac{(X*W_q)(X*W_k)^T}{\sqrt{d_k}})(X*W_v) & \text{for} \quad Attn \\ X*A & \text{for} \quad MLP \end{cases} \tag{2}$$

Note that our row-wise split happens on input's batch dimension, it is mathematically equivalent to vanilla baseline. Given row (batch-dim) split on X mainly affects abstracted computation of $X \otimes A$, we illustrate in details on $X \otimes A$ to show equivalence of our row-split on X and baseline. Element-wise operations like GeLU() and dropout() are completely independent along batch dimension of X, we exclude them for simplicity. Then we get simplified $X \otimes A$ as Equation 2.

For MLP, $X \otimes$ A is just GeMM between X and A. Therefore, as a toy example in Figure 5, row-wise split on X is equivalent to baseline as Equation 3.

$$\begin{bmatrix} X1 \\ X2 \end{bmatrix} * A = X * A \tag{3}$$

For self-attention, we abstract it as $softmax(f(X))g(X)$. For the second part $g(X) = X * W_v$, the equivalence proof here is the same as Equation 3 since it is just a GeMM operation. For $f(X) = \frac{(X*W_q)(X*W_k)^T}{\sqrt{d_k}}$, its output dimensions are $(batch, seq, seq)$. Since Softmax() is conducted on the last dimension of $f(X)$ output as sequence-dim, which is completely independent of first batch dimension. Since $softmax(f(X))$ and $g(X)$ are both independent in batch dimension and their product is also independent in batch dimension, row-wise split on X for self-attention layer is also equivalent to baseline without tensor slicing.

Data dependency: Since the batch dimension of input tensor is completely independent, no synchronization is needed across all transformer layers. As depicted in Figure 5, Domino's row-split on input achieves both *intra-layer* and *inter-layer* (i.e., overlap communication with successive layer computation) computation and communication overlapping.

3.3 Column Split on Weights

With similar analysis as § 3.2, partitioning weights tensor B in row-dimension for N partitions will lead to N^2 times communication volume blow-up. To avoid this, we split the weight tensor B on the column dimension to keep the communication volume the same as the vanilla baseline.

As shown in Figure 6, for B, we split it column-wise to N partitions, and each partial output will have the shape of $(a, d/N)$. After collecting all N chunks, we concatenate these partial results ((e.g., Concat(Y3,Y4) in Figure 6)) at the end of each $X \otimes A \otimes B$ layer computation.

Now we prove column-wise split on weights B is equivalent to baseline without tensor partition. Since dropout() happens after our concatenation as concatenation output is identical to baseline, it can be safely removed from our proof domain. By excluding element-wise dropout() operation, $(XA) \otimes B$ is just GeMM for both self-attention and MLP layers. Thus, the equivalence is shown as Equation 4.

$$(XA) \otimes B = (XA) \otimes [B1, B2] \tag{4}$$

Data Dependency: given this column-wise split on B, for both self-attention layer and MLP layer, the computation output needs to be synchronized at the end of layer execution. As a toy example of column-wise split of 2 shown in Figure 6, Domino achieves *intra-layer* computation communication overlapping but needs synchorize (i.e., Concat(Y3,Y4) in Figure 6) before moving to next self-attention or MLP layer computation.

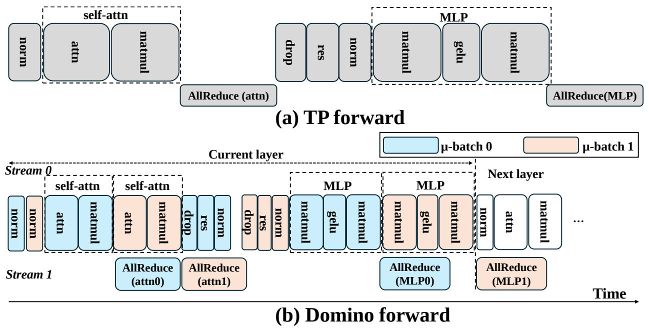


Figure 7: Transformer block (i.e., 1 self-attn and 1 MLP) forward phase. Upper figure is vanilla TP implementation, and bottom figure is Domino implementation.

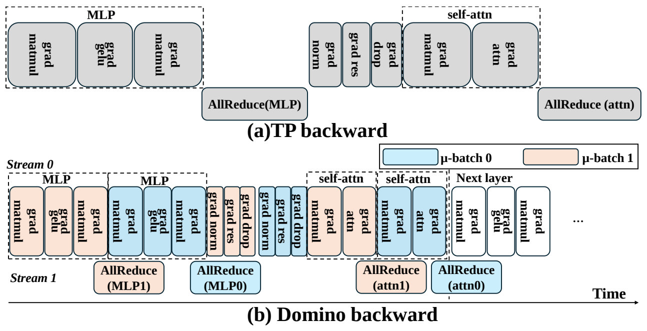


Figure 8: Transformer block (1 self-attn and 1 MLP) backward phase. Upper figure is vanilla TP implementation, and bottom figure is Domino implementation.

3.4 Hybrid Split

For extremely large LLMs [52], we provide a hybrid model split on both input X and last weight tensor B. This hybrid solution is necessary since either row-split or column-split alone would cause narrow shape tensor which is impossible to achieve good computation efficiency. After doing row-wise split on X together with column-wise split on B, Domino can achieve super fine-grained computation and communication overlapping. The aggregated communication size of $X \otimes A \otimes B$ still remains the same as vanilla baseline.

Data Dependency: Inherited from column-wise split on B, for both self-attention layer and MLP layer, final computation outputs need to be synchronized column-wise (i.e., Concat(Y3,Y4) in Figure 6) but non-blocking row-wise. Therefore, the hybrid split can only achieve *intra-layer* computation and communication overlapping.

4 Implementation

We now discuss implementation details, which includes row-wise partitioning strategy for input data (§ 4.1), column-wise partitioning approach for model weights (§ 4.2), and further optimization on computational kernels (§ 4.3).

4.1 Tensor Partitioning on Inputs

We first illustrate the implementation of our novel input partitioning in both forward and backward propagations, separately.

4.1.1 Forward phase

Users can define the number of partitions p_1 for the input, after which the input is divided into p_1 partitions along the batch dimension. A for-loop iterates through each partitioned μ-batch sequentially. Figure 7 depicts the forward phase of a simple example where the layer input is split into two μ-batches (i.e., $p_1 = 2$).

In Figure 7(a), to hide AllReduce communication (i.e., *AllReduce (attn)* in Fig. 7(a)) after the self-attention layer, we first execute the self-attention of μ-batch 0 and then initiate its AllReduce (i.e., *AllReduce(attn0)* in Figure 7(b)) asynchronously to prevent GPU blocking on communication. Subsequently, we immediately proceed self-attention on μ-batch 1. The communication of self-attention of μ-batch 1 (i.e., *AllReduce(attn1)*) overlaps with layer normalization, residual, and dropout operations. The reason for grouping multiple μ-batches' dropout, residual, layerNorm not only enables hiding *AllReduce(attn1) in Figure 7(b)* for forward pass, but also provides proper overlapping space for *Allreduce(MLP0)* in backward pass shown in Figure 8(b).

Similarly to hide *AllReduce(MLP0)* communication in Figure 7(b) in MLP forward, we initiate this AllReduce after MLP computation on μ-batch 0 asynchronously, enabling immediately execute MLP of μ-batch 1 to achieve overlapping. Additionally, *AllReduce(MLP1)* after MLP of μ-batch 1 will overlap with the computation of μ-batch 0 in the successive transformer block.

4.1.2 Backward phase

The corresponding backward is mostly generated by torch.autograd(). Figure 8 shows a toy example of backward pass where the input hidden states are split into two μ-batches ($p_1 = 2$). We carefully organize the execution of gradient computation in these μ-batches to overlap gradient computation and communication.

In Figure 8, we first adopt similar cross μ-batch computation and communication overlapping as described in § 4.1.1. To further broaden overlapping scope, we also adopts overlapping communication with weights gradient computation within the same μ-batch. For example, *AllReduce(MLP1)* in Figure 8(b) partially overlaps with *grad matmul* computation of its own μ-batch 1 (i.e. 3rd orange block from left). Each *grad matmul* usually involves two separate kernel computation as inputs gradient and weights gradient computation. This sub-module overlapping can be achieved by first calculating inputs gradient inside 2nd *grad matmul* in MLP layer of μ-batch 1 (i.e. 3rd orange block from left), and then trigger its weights gradient computation and inputs gradient communication simultaneously.

However, accurate control of gradient communication behavior to overlap with gradient computation is challenging because PyTorch automatically generates the gradient computation graph [69]. To precisely control communication start/end time, our initial attempt to manually implement customized backward pass leads to poor throughput performance due to triggering less efficient kernels than torch.autograd(). To tackle this issue, we developed a *no-operation* module. This module receives the communication handle during the forward phase and retains it for use during the backward phase. Our *no-operation* module integrates seamlessly with torch.autograd(). This approach allows us to precisely control the completion time of asynchronous communication without complicated code modification.

To sum up, *Domino enables up to ∼100% communication hiding behind computation with our batch-split on inputs.*

4.2 Tensor Partitioning on Weights

Users can also define the number of partitions p_2 for weights. Subsequently, p_2 linear modules are initialized, each with hidden dimension scaled by $\frac{1}{p_2}$.

Bottom half of Figure 6 shows a toy example of the weight partition where the number of partitions for weights is 2. Specifically, we first execute the first linear module (i.e., $X \otimes A \otimes B1$) to generate the first half result (i.e., $Y3$). We then trigger asynchronous non-blocking AllReduce on the first half result. After that, we immediately execute the second half linear module ($X \otimes A \otimes B2$). Therefore, *AllReduce(Y3)* is overlapped with $X \otimes A \otimes B2$. In the backward, we adopt similar sub-module overlapping strategy as discussed in § 4.1.2.

An obstacle here is to fully restore hidden dimension (i.e. *concat(Y3,Y4)* in Figure 6) for subsequent operations (e.g., layerNorm, dropout, etc.). torch.cat() often allocates GPU memory more than needed [34], which may trigger unnecessary OOM (out-of-memory) errors. To achieve concatenation on hidden dimension without torch.cat(), we pre-allocate a big buffer to store the first half (i.e., *Y3*) and the second half (i.e., *Y4*) result sequentially in Figure 6. However, this method still incurs extra memory copy (MemCpy) overhead due to non-contiguous memory addresses. We believe this MemCpy overhead can be mitigated or eliminated by implementing customized kernels that simultaneously read from and write to non-contiguous memory addresses. Given current impact of this extra MemCpy is minimal, we defer its optimization to future work.

In practice, Domino achieves up to 50% to 70% communication hiding by employing column-wise split on weights. Although this overlapping percentage is lower than batch-wise input split (§ 4.1), this approach remains essential, since that batch-split alone results in tensors with narrow shapes that hinder kernel computation efficiency.

4.3 Generic Kernel Optimization

Here we discuss generic kernel-level optimizations with CUDA-MultiStream and PyTorch-native compiling techniques.

4.3.1 MultiStream

After splitting the computation into smaller units, the computation required for each kernel is significantly reduced compared to the original TP baseline. To increase GPU utilization while reducing sequential kernel launching overhead, we execute independent operations in parallel using multiple CUDA streams.

To obtain a new CUDA stream, one can retrieve it from the CUDA stream pool. However, this method generates an excessive number of new streams and utilizes them in a round-robin fashion, leading to a high overhead from frequent stream switching. To mitigate this, we first initialize and create a fixed number of global streams before execution. Then, we use an index to obtain a specific stream, thereby reducing the overhead associated with stream switching.

4.3.2 CudaGraph & Torch.compile

torch.compile() functionality from PyTorch accelerates code execution by just-in-time (JIT) compiling PyTorch operations into optimized kernels, enabling improved performance with minimal code modifications [59]. Many operations from the torch library are employed to construct our modules. By fusing distinct operations, we leverage torch.compile() to enhance our computational efficiency.

After Domino slicing tensor into multiple chunks, the computation needed for each chunk is significantly less than the original baseline, leading to GPU idleness between adjacent operations (i.e., bubble time). The primary reason for bubbles is the computation time for different operations is less than the PyTorch scheduling latency. To address this issue, we employ CudaGraph [43, 58] to eliminate the gap between adjacent operations, thereby reducing the overall computation time. However, commonly-used on-device random number generator (RNG) feature is incompatible with CudaGraph. As a workaround, we utilize a fixed seed instead of random numbers to mimic the behavior of the RNG.

5 Evaluation

This section provides detailed evaluation and benchmark results of Domino. We first discuss the model and hardware settings of our experiments (§ 5.1).After that, we describe baseline and evaluation metrics (§ 5.2) Then we evaluate

Model	Model Size	Global Batch Size
GPT-3	2.7B, 6.7B, 13B, 30B	4 - 64
Llama-2	7B, 13B	4 - 64

Table 1: Model type, model size, batch size configurations.

benchmark results on GPT-3 (§ 5.3) and Llama-2 (§ 5.4) models in both single-node and multi-node cases.

5.1 Model and Hardware

Before discussion on detailed evaluation results, we first describe models and hardware settings of our experiments.

5.1.1 Model

We focus on evaluation of GPT [11, 52, 60] and Llama [5, 6, 72, 73] model series. More specifically, we conduct our benchmark tests using GPT-3 [11] model and Llama-2 [73] model with different model sizes. All model configuration details are illustrated in Table 1. For model size calculation, we follow the equation from Nvidia Megatron team [41] as Equation 5, where h refers to the hidden size and l is the number of layers. *seq_len* represents sequence length and *vocab* is vocabulary size.

$$model\_size = (1 + \frac{13}{12*h} + \frac{vocab + seq\_len}{12*h*l}) * 12*l*h^2 \quad (5)$$

5.1.2 Hardware

We conduct experiments on Nvidia DGX-H100 boxes [22], each with 8 H100 GPUs. Within each DGX-H100 node, GPUs are connected with NVLink [50] and NV-Switch [51]. Each DGX-H100 node is equipped with 8 Nvidia InfiniBand ConnectX-7 network cards [42] for cross node communication, which provides an aggregated network bandwidth of 400 GB/s per node. We have three different hardware settings: 1 node, 2 nodes and 4 nodes, which represents both single node and distributed training environments. All nodes run in the same PyTorch environment with NCCL version 2.18 and CUDA version 12.2.

5.2 Metrics

Similar to previous arts on computation-communication overlapping [32, 54, 78], we report results analysis mainly with overall training iteration time. In Equation 6, throughput or TFLOPs can be inferred since it is just inversely proportional to iteration time measurement (i.e., *iter_time*).

$$TFLOPs \propto \frac{1}{iter\_time} \quad (6)$$

We believe iteration time represents more thorough and end-to-end results because CPU side execution (e.g., data pre-processing, learning rate adaptation, etc.) is also taken into account, which may not be included in pure TFLOPs measurements on GPUs. Since we only use TP for model partition and each GPU in TP domain shares the same input, our global batch size is equivalent to our micro batch size.

Our baseline is using a stable release of Megatron-LM with two different settings: synchronous (sync) and asynchronous (async), where sync (i.e., *Megatron-LM(sync)*) means all collective operations are blocking calls and async (i.e., *Megatron-LM(async)*) is to enable backward pass only, coarse-grained computation and communication overlapping feature in Megatron-LM [67]. By default, we compare Domino and *Megatron-LM (async)* with the vanilla *Megatron-LM (sync)* as the baseline.

One thing worth mentioning is that our Domino scheme is mathematically equivalent to vanilla TP solutions like Nvidia Megatron-LM (as proof in § 3.2, 3.3). With fixed random seed and the same learning rate schedule, we monitored through weights & bias tool [1], which shows that Domino's loss curve matches with Megatron-LM baseline. For the sake of brevity, we exclude these convergence results here.

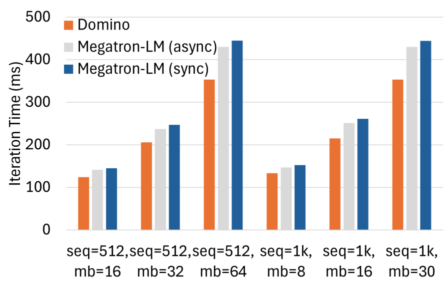
5.3 GPT-3

GPT is popular and representative model series of transformers. We conduct model training benchmark of GPT-3 with different model sizes ranging from 2.7B to 30B using 1 to 4 DGX-H100 nodes (i.e., 8 to 32 H100 GPUs). We use two different sequence lengths 512 and 1024. Given that for row-split (i.e., batch-split) on input, our smallest micro-batch size starts from 4 and up to the maximum batch size without triggering OOM. We exclude the cases of micro-batch sizes 1 and 2. Micro-batch size of 1 is impractical for batch-wise input splitting. Additionally, micro-batch size of 2 is excluded because with minimum half-half split, each half batch size is 1, which is impossible to achieve good training throughput.

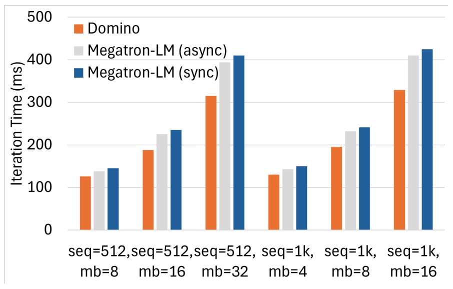
As described in § 3.1, we conduct grid search and only report the best performance numbers of Domino via both row-wise input split and column-wise weights split. Additionally, we also tried to enable or disable features like CudaGraph() and torch.compile() and report the best numbers that Domino achieved. To achieve good throughput and system efficiency, we report benchmark results of top-2/3 largest micro-batch sizes that ≥ 4 and without causing OOM.

5.3.1 Single-node

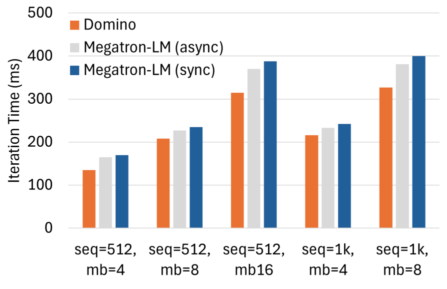
For single node training, we tested model size with 2.7B, 6.7B and 13B. In summary, compared with Megatron-LM, Domino achieves up to 1.3x throughput speed-up. Furthermore, in multiple cases, Domino achieves near-optimal or even exceeds the optimal settings. The optimal solution refers to disabling



(a) GPT-2.7B



(b) GPT-6.7B



(c) GPT-13B

Figure 9: GPT-3 training iteration time with different model sizes, sequence lengths (seq), and micro-batch (mb) sizes on a single DGX-H100 node (8 H100 GPUs).

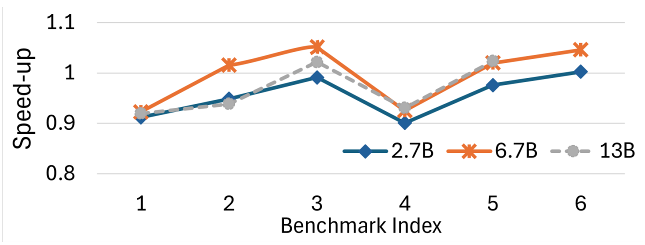


Figure 10: Domino normalized throughput speed-up compared with optimal solution (no communication) on single DGX-H100 (8 H100 GPUs). Benchmark index strictly follows the order of horizontal settings (i.e., seq, mb) in Figure 9.

all the communication in TP training of both forward and backward passes.

One tricky part is whether we should enable CudaGraph or not. Based on our experimental results, we found that if the batch size is small (i.e., training job is not compute-heavy), enabling CudaGraph could squeeze the bubble/idle time between adjacent kernels thus improving end-to-end performance. On the other hand, if the training job is compute-heavy and does not have much idle time between adjacent kernels, we disable CudaGrpah for faster model training initialization and less on-device memory copy overhead. Taking GPT-3 13B training as an example, with sequence length of 512 and micro-batch size of 4, we notice significant training iteration time reduction (around 10-15%) if we switch from cudaGraph off mode to on mode, which shows the benefits of reducing idle time between adjacent compute kernels. On the other hand, if we increase the micro-batch size to 16, enabling cudaGraph leads to 5-10% longer iteration time than disabling it, which is mainly due to extra memory copy overhead introduced in CudaGraph.

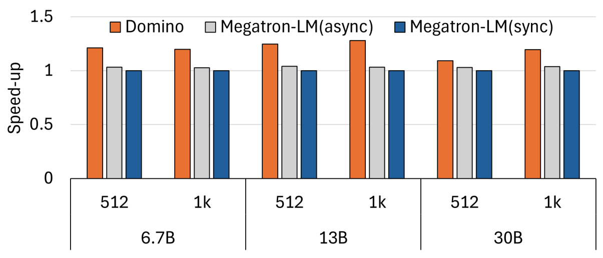
Overall, as shown in Figure 9, enabling Megatron's coarse computation and communication overlapping (i.e., Megatron-LM (async)) achieves around 2-5% throughput gain compared with vanilla megatron baseline. Compared with the Megatron-LM baseline, Domino achieves higher speedup gains when training batch is large and relatively low performance gains for small batch size cases.

For GPT-3 2.7B training shown in Figure 9a, Domino achieves 1.14x to 1.26x speedup over Megatron baseline for both sequence lengths of 512 and 1k. In GPT-3 6.7B training as Figure 9b, since we increase model size from 2.7B to 6.7B, the largest micro-batch sizes are reduced compared with 2.7B cases. However, we achieve the highest throughput gain in 6.7B model compared with 2.7B and 13B cases. More specifically, in Figure 9b, for both sequence lengths of 512 and 1k, we achieve from 1.15x to 1.3x speedup over Megatron baseline with increasing micro batch sizes. For 13B cases in Figure 9c, we have the smallest micro-batch sizes for training, which leads to 12% to 23% throughput speedup over the Megatron baseline with increased batch sizes. In summary, *Domimo generally outperforms Megatron baseline in varied batch sizes and sequence lengths*. Our performance gain increases as the batch size grows.

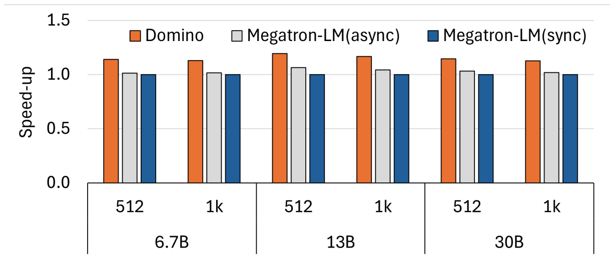
We also depict comparison of Domino performance with optimal settings with different sequence lengths and batch sizes on a single node. As shown in Figure 10, the horizontal benchmark index numbers strictly follow the same order of training settings in Figure 9. Compared with the optimal setting that removes all communications in Megatron-LM, Domino reaches over 90% of optimal throughput in all cases and has a few cases even exceeding the optimal settings. We conduct an ablation study and performance gain breakdown.We find that, for cases where Domino exceeds the optimal setting, the extra performance gain is primarily attributed to our kernel-side optimization as discussed in § 4.3.

5.3.2 Multi-node

Compared with single node results, multi-node cases are different given cross-node IB bandwidth is still 2-3x lower compared with intra-node NVLink/NVSwitch. Therefore, it is still possible that a single NCCL collective can be longer than the maximum number of computation kernels that Domino can overlap with.



(a) 2 DGX-H100 (16-H100 GPUs)



(b) 4 DGX-H100 (32-H100 GPUs)

Figure 11: GPT-3 training throughput speedup with different model sizes, sequence lengths (seq) and micro-batch (mb) sizes on 2 (16-H100) and 4 DGX-H100 nodes (32-H100 GPUs).

For 2 and 4 DGX-H100 nodes experiments, we test three different model sizes as 6.7B, 13B and 30B across 16 to 32 H100 GPUs with TP model partition strategy. As shown in Fig. 11, we report normalized throughput speed-up when comparing Domino with Megatron baseline. For both sequence lengths of 512 and 1k, we present our best throughput results with proper batch sizes ranging from 4 to 64. Coarse-grained computation and communication overlapping provided by Megatron-LM (i.e., Megatron-LM (async) in Fig. 11) gives around 2%-4% performance gain on average.

As shown in Fig. 11a, for 2-node case (16 H100 GPUs), Domino achieves around an average of 1.2x speedup over Megatron baseline for both 6.7B and 30B models with varied sequence lengths and batch sizes. More interestingly, for 13B training, Domino achieves up to 1.3x throughput gain over baseline on 1k sequence length. We believe GPT-3 13B training on 2 DGX-H100 nodes provides a sweet spot that 1) most computation kernels are still highly efficient after our row-wise and column-wise split on inputs and weights, 2) cross-node communication can be mostly overlapped with computation using Domino.

For 4-node case depicted in Fig. 11b, Domino achieves 1.14x to 1.2x throughput speedup over Megatron baseline across GPT-3-6.7B, GPT-3-13B, GPT-3-30B across different batch sizes and sequence lengths. The reason for less performance gain compared with 2-node cases is cross-node communication cannot be perfectly overlapped with the maximum range of computation kernels of Domino. Given the latest IB with Nvidia ConnectX-8 cards [48] could provide 800 GB/s inter-node communication bandwidth, we did a simulation projection with 800 GB/s cross-node bandwidth for both Megatron-LM and Domino. In simulation, Domino could achieve up to 1.5x speedup over Megatron baseline. In our simulation, we also note that Domino could potentially achieve higher speedup over Megatron baseline on larger scales (e.g., 128, 256 GPUs).

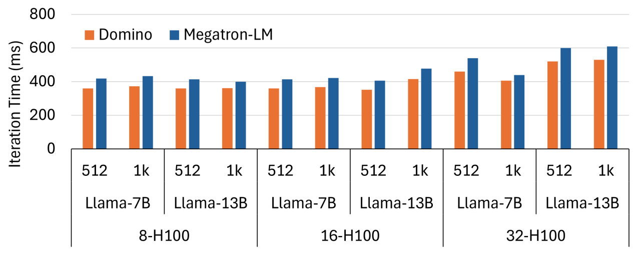


Figure 12: Llama-2 training iteration time with different sequence lengths and model sizes on 1 DGX-H100 node (8-H100), 2 nodes (16-H100), and 4 nodes (32-H100) cases.

5.4 Llama-2

We describe our evaluation of Llama-2 model. The major difference between GPT-3 and Llama-2 model is Llama-2 involves new normalization as Root Mean Square Normalization (RMSNorm) [80], new activation function SwiGLU (SwiGated Linear Unit) [66] and RoPE (Rotary Position Embedding) [68]. Given that the latest Llama-3 [6] model shares a similar model architecture. Compared with Llama-2, some major changes of Llama-3 are RoPE [68] size, hidden size, and embedding configuration difference. Therefore, our results on Llama-2 can be representative of the Llama model series. To avoid duplicated results patterns such as § 5.3, we only report the results of the largest batch size without triggering OOM. Since Megatron with its coarse computation communication overlapping show similar performance as vanilla baseline (only 2% to 4% gain on average), we exclude the Megatron-LM (async) result here. Similar to § 5.3, we also benchmark two different sequence lengths as 512 and 1024 (i.e., 1k in Fig. 12).

5.4.1 Single-node

For single node experiments as shown in left-most 8 bars under 8-H100 column in Figure 12, Domino achieves around 1.16x speedup for Llama-7B training, and 1.1 to 1.15x speedup for Llama-13B training. The lower performance gain on larger model is because we can support smaller batch

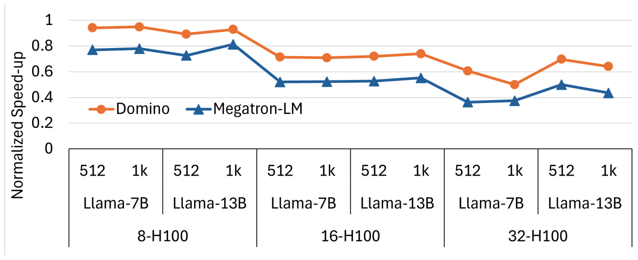


Figure 13: Normalized speedup when comparing Domino, Megatron with Optimal setting (i.e., no comm.) for Llama-2 training on 1 DGX-H100 node (8-H100), 2 nodes (16-H100) and 4 nodes (32-H100) cases.

size training. With smaller batches, Domino's kernel launching overhead becomes more noticeable thus leading to less throughput gain.

Compared with results from GPT-3, Domino has less performance gain over Megatron-LM. The main issue is due to the rotary embedding feature introduced in Llama-2 model. This rotary embedding builds extra data dependency among our input batch dimension split pieces. For better system performance, we leave this rotary embedding issue as a future optimization direction.

We also compare both Domino and Megatron-LM with optimal throughput scenarios (i.e., no communication). As shown in Figure 13, the left most 4 groups of data points under 8-H100 column present Domino and Megatron-LM throughput performance. Here we normalize optimal throughput as 1 and calculate the corresponding throughput of both Domino and Megatron. For Domino in 8-H100 cases, we mostly achieve around 90+% optimal throughput and 10% better than Megatron, which is quite decent.

5.4.2 Multi-node

Similar to single node training, we benchmark Llama-2 7B and 13B training on 2 and 4 nodes with sequence lengths of 512 and 1k.

In Figure 12, two node cases are the middle 8 bars under 16-H100 column. Compared with Megatron-LM, Domino achieves around 1.15x speedup for both 512 and 1k sequence lengths. For 4 nodes case, similar results are shown as the right-most 8 bars under 32-H100 column of Figure 12. Domino achieves 1.08x to 1.17x speedup over Megatron-LM in various model sizes and sequence lengths of 4-node cases.

When comparing to the optimal case (i.e., no communication), as shown in Fig. 13, for both 2-node and 4-node cases, Domino achieves around 60-80% of optimal throughput and consistently around 10-20% better than Megatron-LM.

6 Related Work

Previous literature on reducing communication overhead in distributed model training mainly falls into two categories: overlapping communication with computation, and optimization on collective communication.

6.1 Overlapping Communication with Computation

One major line of overlapping communication with computation is to provide a better scheduling policy. Centauri [13] is a recent work on overlapping communication and computation for hybrid parallelism (e.g. DP, TP, PP, SP) scenarios. Its multi-level partitioning and scheduling introduce significant planning overhead. Additionally, the generated schedule is complex which makes it hard for end-to-end correctness debugging, thus making the proposed scheme less practical. Furthermore, adopting hierarchical collective (e.g., all-gather) does not reduce the overall cross-node communication volumes, and hierarchical collective calls (first intra-node then inter-node) could lead to longer end-to-end network latency in practice. Alpa [83] conducts compiler-level optimization on intra and inter-operator parallelism as well as better overlapping with communication. Similar as Centauri, Alpa's tensor partition strategy is complicated and at compiler-level, which makes it almost impossible to conduct correctness debugging on the user side. Different from Alpa, Domino is mainly at kernel scheduler-level and our solution is clean and neat for correctness debugging. Furthermore, Alpa achieves similar throughput as Megatron-LM while Domino outperforms Megatron-LM. Lancet [32] leverages unique feature of MoE (Mixture-of-Experts) model and overlap All-to-all collectives with forward and backward computation, which is orthogonal to Domino as we focus on dense model type as it is more widely used (e.g. Llama [5, 6, 72, 73], GPT [11, 52], Phi [3] model series). TicTac [25] provides near-optimal computation communication overlapping in Parameter Server (PS) [37] architecture. However, TicTac approach cannot be applied in LLM training since modern large model training only uses MPI-based architecture (i.e. all-worker) [8, 39] rather than PS. Breadth-first pipeline-parallism [35] partitions model layers to GPUs in a round-robin [77] fashion, which interleaves computation with communication and mitigates the burst of communication calls in vanilla pipeline parallel training. This optimized pipeline parallelism is beneficial for low bandwidth interconnect scenarios. However, it may have minor performance gain for state-of-the-art HPC (High-Performance Computing) clusters equipped with high bandwidth InfiniBand [27] links as the scenarios that Domino focuses on.

Another main line of work is kernel fusion of computation and communication. Researchers from Google [78] focus on intra-layer overlapping via kernel fusion of GeMM with collective operations on TPUs. T3 [54] and Flux [12] apply and extend similar ideas on Nvidia GPUs. However, these kernel fusion works only overlap collectively with one specific compute kernel type (i.e., GeMM), which limits its overlapping scope. CoCoNet [29] provides a general kernel fusion paradigm that automatically generates fused kernel between collective and popular compute operations (i.e., GeMM and convolution). However, the generated code achieves less system efficiency compared with directly using highly-optimized kernels from cuBlas [15], cutlass [17] or cuDNN [16] mainly due to extra in-kernel synchronization introduced for this fine-grained compute-communication overlapping. MGG [79] fuses computation and communication kernels for graph neural network (GNN) via NVSHMEM [49]. Researchers from AMD [57] also fuse embedding and GeMM with collectives to achieve fine-grained compute-communication overlapping on AMD MI200-series GPUs. Compared with these GeMM+NCCL fusion work, Domino provides more flexible and wider range of computation and communication overlapping. Furthermore, Domino is orthogonal to this compute-communication kernel fusion line of work, which can be adopted to further improve Domino system efficiency.

6.2 Fast Collective Communication

Collective communication libraries like NCCL [30], Gloo [4], Blink [76], Horovod [64], optimize collective communication itself to reduce communication overheads in distributed model training. For example, NCCL [30] incorporates Infiniband Sharp technology [28] and its CollNet optimization [31] for in-network data aggregation to reduce communication volume as well as tensor reduction overhead. ACE [63] is a similar work of offloading collectives into network fabric from academia, which provides good simulation numbers. Gloo [4] provides general collective primitive supports on CPU side. Horovod [64] reduces communication overhead by batching multiple collective calls, thus reducing kernel launching overheads. Blink [76] improves network utilization by providing a spanning tree communication protocol that leverages idle links that cannot form ring topology. ByteScheduler [56] incorporates both parameter server architecture [37] and MPI all-worker architecture [9, 39, 71] for hybrid collective primitive design, and switches between these two schemes for better network utilization. MSCCL [14] and its following work [40, 65] optimize collective communication via various technologies such as compiler optimization [14, 40], sketch abstraction [65], etc.

Domino is orthogonal to all collective optimizations and can plug in any collective library if needed. We choose NCCL by default given its wide adoption in distributed model training on Nvidia GPUs [21, 22]. To enable Domino to run on AMD GPUs, simply replacing NCCL calls with corresponding RCCL [7] collectives from AMD would work seamlessly.

7 Conclusion

We propose Domino, a generic approach for tensor slicing and partitioning to achieve fine-grained overlapping of computation kernel sequences with communication collectives. Extensive results on multiple DGX-H100 nodes show that, Domino can achieve up to 1.3x speedup over the state-of-the-art tensor parallelism solution as Megatron-LM. Furthermore, Domino even exceeds optimal performance (i.e., remove all communication in Megatron-LM) in some cases. With the trend of high communication bandwidth and faster computation per accelerator, Domino could be beneficial for both small scale and large-scale LLMs training.

Acknowledgments

We would like to thank Shiyang Chen (Rutgers University), Shuaiwen Leon Song (together.ai), Hexu Zhao (NYU) and Yuxiong He (Snowflake) for valuable feedbacks and code contributions to our Domino project.

Availability

Domino is an open source project from Microsoft DeepSpeed Team (`https://www.deepspeed.ai/`). Code implementation will be publicly available at `https://github.com/microsoft/DeepSpeed`

References

[1] Weights & Biases. https://wandb.ai/site, 2024.

[2] PyTorch 2.3. Tensor parallelism introduces more efficient ways to train llms. https://tinyurl.com/3fjvd88t, 2024.

[3] Marah Abdin, Sam Ade, Ammar Ahmad, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sebastien Bubeck, Martin Cai, Caio Cesar Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Parul Chopra, Allie Del Giorno, Gustavo de Rosa, Matthew Dixon, Ronen Eldan, Dan Iter, Amit Garg, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Jamie Huynh, Mojan Javaheripi, Xin Jin, Piero Kauffmann, Nikos Karampatziakis, Dongwoo Kim, Mahoud Khademi, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Chen Liang, Weishung Liu, Eric Lin, Zeqi Lin, Piyush Madan, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Xia Song, Masahiro Tanaka, Xin Wang, Rachel Ward, Guanhua Wang, Philipp Witte, Michael Wyatt, Can Xu, Jiahang Xu, Sonali Yadav, Fan Yang, Ziyi Yang, Donghan Yu, Chengruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[4] Meta AI. Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, 2024.

[5] Meta AI. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024.

[6] Meta AI. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/, 2024.

[7] AMD. ROCm Collective Communication Library (RCCL). https://rocm.docs.amd.com/projects/rccl/en/latest/index.html, 2024.

[8] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml. In *MLSys*, 2022.

[9] Blaise Barney. Message Passing Interface. https://computing.llnl.gov/tutorials/mpi/, 2018.

[10] Iz Beltagy, Matthew Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. Flux: Fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[13] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling Efficient Scheduling for Communication-Computation Overlap in Large Model Training via Communication Partitioning. In *ACM ASPLOS*, 2024.

[14] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. MSCCLang: Microsoft Collective Communication Language. In *ACM ASPLOS*, 2023.

[15] cuBLAS library. https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf, 2024.

[16] cuDNN: NVIDIA CUDA Deep Neural Network library. https://developer.nvidia.com/cudnn, 2024.

[17] cutlass: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass, 2024.

[18] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *ACM/IEEE SC*, 2005.

[19] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[20] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[21] Introduction to the NVIDIA DGX A100 System. https://docs.nvidia.com/dgx/dgxa100-user-guide/introduction-to-dgxa100.html, 2024.

[22] NVIDIA DGX H100 Datasheet. https://resources.nvidia.com/en-us-dgx-systems/ai-enterprise-dgx, 2023.

[23] Tobias Gysi, Jeremia Bar, and Torsten Hoefler. dcuda: Hardware supported overlap of computation and communication. In *ACM/IEEE SC*, 2016.

[24] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[25] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *MLSys*, 2019.

[26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, 2019.

[27] Introduction to InfiniBand. https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf, 2007.

[28] InfiniBand In-Network Computing With NVIDIA SHARP. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/network-computing-nvidia-sharp, 2021.

[29] 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 *ACM ASPLOS*, 2022.

[30] Sylvain Jeaugey. Optimized inter-GPU collective operations with NCCL 2. https://developer.nvidia.com/nccl, 2017.

[31] Sylvain Jeaugey. NCCL Collnet, CollnetDirect and CollnetChain. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-algo, 2024.

[32] Chenyu Jiang, Ye Tian, Zhen Jia, Shuai Zheng, Chuan Wu, and Yida Wang. Lancet: Accelerating Mixture-of-Experts Training via Whole Graph Computation-Communication Overlapping. In *MLSys*, 2024.

[33] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *MLSys*, 2023.

[34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *ACM SOSP*, 2023.

[35] Joel Lamy-Poirier. Breadth-First Pipeline Parallelism. In *MLSys*, 2023.

[36] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *ICLR*, 2021.

[37] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*, 2014.

[38] Vladimir Marjanovic, Jesus Labarta, Eduard Ayguade, and Mateo Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010.

[39] Mpi: A message passing interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993.

[40] MSCCL++: A GPU-driven communication stack for scalable AI applications. https://github.com/microsoft/mscclpp, 2024.

[41] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunt, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. fficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[42] Nvidia. NVIDIA CONNECTX-7. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf, 2021.

[43] Nvidia. Constructing CUDA Graphs with Dynamic Parameters. https://developer.nvidia.com/blog/constructing-cuda-graphs-with-dynamic-parameters/, 2022.

[44] Nvidia. A PyTorch Extension: Tools for easy mixed precision and distributed training in Pytorch. https://github.com/NVIDIA/apex, 2024.

[45] Nvidia. Dgx-b200. https://www.nvidia.com/en-us/data-center/dgx-b200/, 2024.

[46] Nvidia. Introduction to the NVIDIA DGX H100 System. https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html, 2024.

[47] Nvidia. Megatron-lm release ver 23.08. https://github.com/NVIDIA/Megatron-LM/tree/23.08, 2024.

[48] Nvidia. NVIDIA CONNECTX-8. https://www.nvidia.com/en-us/networking/products/infiniband/quantum-x800/, 2024.

[49] Nvidia. NVIDIA OpenSHMEM Library (NVSHMEM). https://developer.nvidia.com/nvshmem, 2024.

[50] NVIDIA NVLINK. http://www.nvidia.com/object/nvlink.html, 2017.

[51] NVIDIA NVSWITCH. http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, 2018.

[52] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[53] OpenAI. ChatGPT. https://chatgpt.com/, 2024.

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

[55] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. Tale of two cs: Computation vs. communication scaling for future transformers on future hardware. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2023.

[56] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *ACM SOSP*, 2019.

[57] Kishore Punniyamurthy, Khaled Hamidouche, and Bradford M. Beckmann. Optimizing distributed ml communication with fused computation-collective operations. *arXiv preprint arXiv:2305.06942*, 2023.

[58] Meta PyTorch. Accelerating PyTorch with CUDA Graphs. https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/, 2021.

[59] Meta PyTorch. torch.compile. https://pytorch.org/docs/stable/generated/torch.compile.html, 2024.

[60] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf, 2019.

[61] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *ACM/IEEE SuperComputing*, 2020.

[62] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

[63] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. Enabling compute-communication overlap in distributed deep learning training platforms. In *ACM/IEEE ISCA*, 2021.

[64] Alex Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[65] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *USENIX NSDI*, 2023.

[66] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[67] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[68] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[69] Meta PyTorch team. PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/pytorch, 2024.

[70] Meta PyTorch team. Tensor parallelism okr in pytorch distributed h2 2024 roadmap. https://drive.google.com/file/d/19cim5wCoxf8A66YZzoMrjardRntpX6gH/view, 2024.

[71] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19, 2005.

[72] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothee Lacroix, Baptiste Roziere, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[73] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[74] vLLM. Distributed Inference and Serving with TP. https://docs.vllm.ai/en/stable/serving/distributed_serving.html, 2024.

[75] vLLM team. A high-throughput and memory-efficient inference and serving engine for LLMs. https://github.com/vllm-project/vllm, 2024.

[76] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*, 2020.

[77] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In *MLSys*, 2021.

[78] 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 *ACM ASPLOS*, 2023.

[79] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. Mgg: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-gpu platforms. In *USENIX OSDI*, 2023.

[80] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[81] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16, 2023.

[82] Yihao Zhao, Yuanqiang Liu, Yanghua Pen, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *ACM SIGCOMM*, 2022.

[83] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *USENIX OSDI*, 2022.