FOLD3D: Rethinking and Parallelizing Computational and Communicational Tasks in the Training of Large DNN Models

Fanxin Li, Shixiong Zhao*, Yuhao Qing, Xusheng Chen,
Xiuxian Guan, Sen Wang, Gong Zhang, Heming Cui

Abstract—Training a large DNN (e.g., GPT3) efficiently on commodity clouds is challenging even with the latest 3D parallel training systems (e.g., Megatron v3.0). In particular, along the pipeline parallelism dimension, computational tasks that produce a whole DNN's gradients with multiple input batches should be concurrently activated; along the data parallelism dimension, a set of heavy-weight communications (for aggregating the accumulated outputs of computational tasks) is *inevitably serialized* after the pipelined tasks, undermining the training performance (e.g., in Megatron, data parallelism caused all GPUs idle for over 44% of the training time) over commodity cloud networks. To deserialize these communicational and computational tasks, we propose the AIAO scheduling (for 3D parallelism) which slices a DNN into multiple segments, so that the computational tasks processing the same DNN segment can be scheduled together, and the communicational tasks that synchronize this segment can be launched and overlapped (deserialized) with other segments' computational tasks. We realized this idea in our FOLD3D training system. Extensive evaluation shows FOLD3D eliminated most of the all-GPU 44% idle time in Megatron (caused by data parallelism), leading to 25.2%-42.1% training throughput improvement compared to four notable baselines over various settings; FOLD3D's high performance scaled to many GPUs.

Index Terms—Deep Learning, distributed training, GPU, DNN, 3D parallelism, pipeline parallelism, Machine Learning

1 INTRODUCTION

THE high modeling capacities of a large DNN (e.g., GPT3 [1] with 175 billion parameters) have made training or fine-tuning (essentially, training) such a model prevalent and frequent on commodity clouds. Various cloud tenants, including small enterprises, research labs, and individual researchers, frequently train or fine-tune such a large DNN for broad applications [2], [3], [4], [5], [6], [7] with their own private datasets and application needs. The working-set memory (i.e., in-GPU memory, without further specified) needed for training the model far exceeds the capacities of individual accelerators (e.g., GPUs), flourishing parallel techniques that split a DNN model across devices.

3D parallelism [8], [9] (Figure 1a) is a crucial DNN training technique that combines and orchestrates three parallelism dimensions. Tensor parallelism (TP) splits a single DNN operator (often too large to fit in one device) over devices. Pipeline parallelism (PP) [10], [11] places different operator sets (i.e., pipeline stages) of a DNN model over devices and pipelines the execution of multiple micro-batches (i.e., splits of a single SGD batch which is a set of training inputs for each SGD [12] iteration) to reduce devices' idling time, as Figure 4 shows. Data parallelism (DP) replicates the model across devices, lets each replica handle one micro-batch, and synchronizes the gradients produced by all micro-batches after finishing one SGD batch [13].

Overall, the end-to-end performance of 3D parallel training can be divided into two runtime phases: a *configuration phase* and a *scheduling phase*. First, given a DNN model and an AI cluster of N GPU devices connected by hierarchical inter-links (e.g., NVLink [14] within a host and RDMA [15] across hosts), the configuration phase determines the number of splits in TP t, the number of splits in PP p, and the number of DP replicas d, where $t * p * d = N$. Second, given the above 3D configuration, the scheduling phase determines the order in which the devices actually execute the computation tasks of each micro-batch and communication tasks between devices (TP.sync, PP.sync, and DP.sync in Figure 1a). The two phases collectively decide the effective total GPU ALU utilization, under the bounds of per-GPU memory and the heterogenous inter-links.

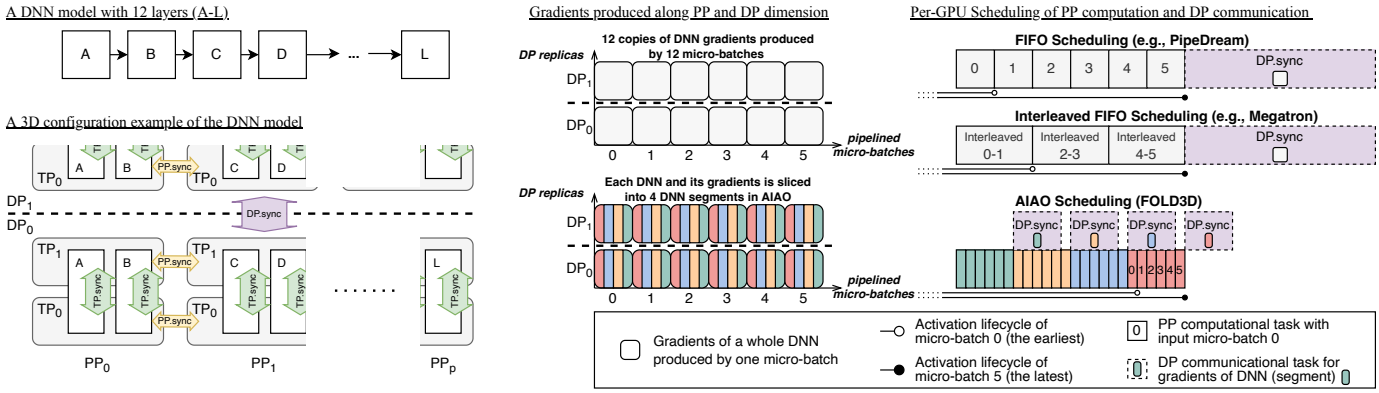
Many recent works [8], [16], [17] focus on finding an optimal 3D configuration. For example, to place DNN models that are too large to fit in one device, while TP and PP both fit for splitting a model, Megatron-PTD [8] and Piper [16] prefer TP over PP with the existence of fast inter-links such as NVLink (often available within a host), as TP often achieves higher computational efficiency [8] in such cases. Inversely, for inter-links such as RDMA and Ethernet, PP is favored [8], [16]. Specifically, the RDMA or Ethernet in the most high-end commodity cloud (e.g., AWS) is up to 400Gbps for the entire cluster. Moreover, the networks in the same commodity cloud are shared by many tenants. Even for the top-of-the-line AWS cloud, the network bandwidth for a single tenant is often merely up to 70 or 80 Gbps (confirmed in §6).

Unfortunately, despite much effort in optimizing the configuration phase, in the scheduling phase, existing 3D training systems are inevitably trapped in a serialization problem, where heavy communication blocks the computation and causes devices idling. Specifically, as shown in

* *Shixiong Zhao is the corresponding author.*

- *Fanxin Li, Shixiong Zhao, Yuhao Qing, Xusheng Chen, and Xiuxian Guan are with the Department of Computer Science, The University of Hong Kong, HKSAR, China. E-mail: {fxli, sxzhao, yhqing, xschen, xxguan}@cs.hku.hk.*
- *Sen Wang and Gong Zhang are with the Theory Lab, 2012 Labs, Huawei Technoloies, Co. Ltd, HKSAR, China. E-mail: {wangsen31, nicholas.zhang}@huawei.com.*
- *Heming Cui is with the Department of Computer Science, The University of Hong Kong, HKSAR, China, and also with Pujiang Lab, Shanghai, China. E-mail: heming@cs.hku.hk.*

Manuscript received 13 Jul 2022; revised 22 Nov 2022 and 29 Jan 2023; accepted 14 Feb 2023.



a) 3D Parallelism

b) Per-GPU Scheduling with Forward Pass Omitted

Fig. 1: a) 3D Parallelism. Each gray box is a GPU device. Sync stands for the communications that synchronize each parallelism dimension. b) A conceptual illustration of the serialization problem and our idea. The gradient computational tasks are represented by backward passes of DNN training (with forward passes omitted and full scheduling being shown in Figure 4). In AIAO scheduling, a copy of DNN gradients produced by micro-batch i is sliced into four segments (with distinct colors), and the same segments (i.e., the same colored ones) are grouped together during pipelining. Compared with the FIFO-based scheduling, our AIAO scheduling moves the DP.sync tasks off the performance critical path by introducing larger total lifecycles of activation checkpoints and resulting in a larger peak GPU memory.

Figure 1b, the gradients of a whole DNN in 3D parallel training are computed as follows: (1) along the PP dimension, on each device (i.e., pipeline stage), a DNN's gradient copies are computed by pipelining multiple micro-batches (six micro-batches in Figure 1b) and accumulated locally on each device; (2) along the DP dimension, the accumulated gradients of the whole DNN produced by different DP replicas (two replicas in Figure 1b) are synchronized across devices by a heavyweight all-reduce [18] communication (DP.sync tasks).

However, in existing 3D parallel systems [8], [16], [17], [19], the micro-batches along the pipeline dimension are usually scheduled with a First-In-First-Out (FIFO) order in terms of micro-batch ID: a later enqueued task (with larger micro-batch ID) should not start until an earlier enqueued task (with smaller micro-batch ID) finish. Meantime, the launch of the DP.sync communicational tasks which synchronize the gradients produced by the pipelined micro-batches across DP replicas must be serialized after the accomplishment of the last pipelined computational task (i.e., micro-batch 5 in Figure 1b).

Theoretically, most conventional (pure) pipeline parallel training systems [10], [11], [20], [21], [22] adopt the FIFO scheduling principle so as to minimize the average lifecycles of the computational tasks' working-set memory residing in the limited and expensive GPU memory [23]. 3D parallel training systems inherit this principle (e.g., Megatron [8] and Alpa [17] all select a FIFO-based scheduling, named 1F1B), since even with a bunch of memory squeezing techniques (e.g., activation checkpointing [8], [22]), GPU memory is still a major bound for 3D parallel training scaling large [16], [19], [20], [24], [25]. Nevertheless, because of the serialization problem, the performance of these 3D parallel training systems is inevitably capped by the sum of computational and communicational costs (see §3.3).

Empirically, in the four most notable baseline systems we extensively evaluated (e.g., Megatron v3.0 [26], the latest 3D parallel training system released by Nvidia in May 2022), the serialization problem makes the DP.sync communicational tasks heavily block the computational tasks of the next training step and causes GPUs to idle for up to 44% of the total training time, over a 256 A100 cluster with 200Gbps inter-host links (an extremely private cluster setting that the entire cluster is used by us only, see §6). When using Megatron to train a GPT3-18B model over a 200Gbps cloud network, the per-GPU hardware utilization is merely 67.4 TFLOPs (FOLD3D achieved 95.8 TFLOPS).

In this paper, we argue that *inheriting the best scheduling of each individual parallelism dimension does not necessarily result in the best holistic scheduling for 3D parallelism on commodity clouds.* Instead, by relaxing the optimality of scheduling in the PP dimension, we can achieve a better scheduling for 3D parallelism that greatly alleviates the serialization problem, leveraging two subtle observations (depicted in Figure 1b).

The first observation is that, although there inevitably exists a serialized dependency between the PP computational tasks (with each processing the *whole DNN's* gradients from a micro-batch) and the DP communicational tasks (which synchronize the whole DNN's gradients with other DP replicas), we can slice a DNN model into segments (conceptually, sub-DNNs) with each containing distinct consecutive DNN layers, so that both the PP computational and DP communicational tasks can be divided into sub-tasks (per sub-DNN/segment). By doing so, although within each sub-task, the serialization dependencies still exist, we can schedule (group) subtasks that process the same segment's gradients (e.g., the gradients in green in Figure 1b) from all micro-batches together, so that the DP.sync tasks for this part's gradients (e.g., DP.sync for green gradients) can be immediately launched and overlapped with the computational tasks of the other segments (e.g., yellow computations).

We realized this observation in the All-In-All-Out (AIAO) scheduling (Figure 1b), a holistic 3D parallel scheduling, where the scheduling of computational tasks does not depend on input (micro-batch) IDs but is based on the segment (sub-DNN) dependencies. Thus, from the view

of input micro-batches, our AIAO essentially needs all input micro-batches (all-in) enqueued, so as to allow the grouped scheduling of segment sub-tasks corresponding to these micro-batches. The full scheduling depicted in Figure 4 and §3.2 shows that within each segment of the model, our AIAO separately groups the forward pass and backward pass of the pipelined computational tasks on this segment and schedules the grouped tasks according to segments' forward-backward dependencies. To alleviate serialization, a segment's DP.sync task is scheduled to be overlapped with both the backward and forward pass computational tasks in a mirrored way.

However, one challenge for the above AIAO scheduling is that, altering the FIFO principle would inevitably introduce longer activation lifecycles in total and increase the peak memory usage of GPUs (§3.3).

The second observation to address this challenge is that, there exists an invariant architectural opportunity for any pipeline schedule, where all micro-batches share the same size of computation window (the sum of one micro-batch's forward pass and backward pass) which allows offloading [27] critical activation checkpoints to the host memory, despite the order of micro-batches enqueueing and dequeueing. By doing so, the increased memory burden is shifted from GPUs to hosts, as the CPU memory on a host is orders of magnitude larger than the memory capacity of each GPU (§2.1). Leveraging this observation, the AIAO scheduling is accompanied with two key memory squeezing mechanisms (§4): an intra-segment offloading mechanism and an inter-segment lazy communication mechanism, making the AIAO scheduling incur negligible extra GPU memory burden when training large DNNs (Table 2).

We implemented the AIAO scheduling in FOLD3D based on Megatron [26], a well-engineered and open source 3D training system, by adding 5371 LoC. We compared FOLD3D against Megatron-SP [28] (v3.0.2, the latest release), Megatron-PTD [8] (v2.5.0), DeepSpeed Zero3 [29] (DSpeedZ3), and DeepSpeed 3D [19] (DSpeed3D), covering three notable and open source 3D training systems and one state-of-the-art data parallel training system (DSpeedZ3). Both FOLD3D and Megatron-SP are enabled with sequence parallelism (§5), one of the latest memory squeezing techniques [28] complementary to 3D parallelism. Our evaluation was done both over a high-profile cluster (256 A100 GPUs) and a middle-profile cluster (64 V100 GPUs). The numbers of GPUs evaluated are comparable to the latest works [16], [17] that study 3D training. We evaluated all five notable large Transformer [30] based models [1], [29], [31], [32], [33] evaluated by recent systems [8], [16], [29]. The extensive evaluation shows that:

- FOLD3D is high-performance on commodity cloud networks. FOLD3D achieved 25.2%-42.1% higher throughput than the baselines with all systems being deployed on both the A100 cluster and the V100 cluster.
- FOLD3D's high performance is robust. By setting various stringent model shapes (e.g., a slip model with a large layer number and a small layer size), FOLD3D's high performance was consistently observed.
- FOLD3D is scalable. Our scalability evaluation over 256 A100 GPUs shows that FOLD3D's performance gain over Megatron was stable (∼31%) from 64 GPUs to 256 GPUs with the model's scale increased correspondingly (i.e., weak scaling, see Figure 2). When each tenant trains a GPT-3 instance, FOLD3D can save the tenant's electricity for about 100,000 KWh over 256 A100 GPUs, which is roughly equal to the electricity used by 100 families per year or tens of electric cars' lifetime (§6.5).
- The increase (relaxing) of FOLD3D's memory consumption is moderate. Table 2 shows that for each host with eight A100 GPUs, FOLD3D consumed in total 8.1GB-17.3GB extra CPU memory, while FOLD3D's GPU memory usage was comparable to baselines'.

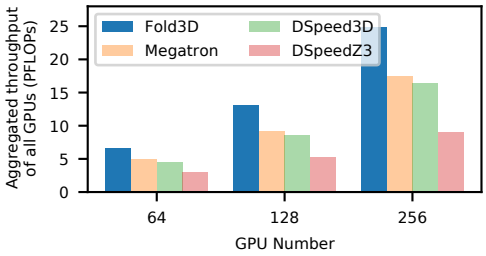


Fig. 2: Weak Scaling of FOLD3D on different amounts of GPUs. FOLD3D consistently achieves high TFLOPs per GPU under various GPU numbers. We used GPT-3 18B, GP-3 39B, GPT-3 81B for each amount of GPUs.

Our contributions are as follows. We take the first step to systematically summarize (Figure 1b) and quantitatively model (§3.3) the serialization problem in existing 3D training systems. We propose the idea of folding, design the AIAO scheduling, and practically realize it in FOLD3D. Leveraging these contributions, we maximally overlap computation and communication tasks in 3D parallel training. FOLD3D can greatly promote many more researchers and enterprises to enjoy the benefit of training and fine-tuning large DNN models on commodity clouds. We believe FOLD3D can benefit various emerging large DNN paradigms such as Mixture-Of-Experts (MoE) [34], [35], Pathways Language Model (PaLM) [36] and Multi-Modal Learning [37], because 3D parallelism is the foundation for these paradigms to scale large; meantime, we envision that it would be challenging to fuse the AIAO scheduling with these new paradigms (§7), and we leave this in future work. Our code and evaluation results are released at github.com/hku-systems/fold3d.

2 Background and Motivation

Existing large models with billions of parameters trained on 3D parallel training systems are mainly stacked up with homogeneous blocks (e.g., transformer block). The repeated structure of these models is their fundamental advantage to obtain better model capacity (and thus higher accuracy) by simply scaling up the model size [30], [31]. In this paper, same as Megatron [8], we assume all models are repeatedly stacked transformer models.

2.1 Parallelism Dimensions

ML model training proceeds with iterations of forward and backward pass computations on micro-batches of a dataset. However, fitting existing large models into a single

GPU for training is unrealistic [25], which expedites the development of parallel training systems that cope with two crucial requirements. First, a system should fit the model's parameters and intermediate results (e.g., activation maps) into a GPU's memory. Even on the top-of-the-line AWS AI clusters, each GPU's memory is limited (e.g., an A100 GPU has 40GB or 80GB memory), while the CPU memory on a host is orders of magnitude larger (e.g., Terabytes) and much cheaper than a GPU's memory. Second, a system should be capable to scale up the training to more GPUs. Certainly, all these two requirements should be met as efficiently (more effective FLOPS per GPU) as possible.

Data Parallelism (DP). In data parallelism [38], each worker has a copy of the model, and the dataset is split across workers. The workers synchronize their gradients periodically via an all-reduce [18] communication (i.e., DP.sync) to maintain a consistent version of the parameters. For a large model which does not fit in a single worker, although pure DP practice [19], [39], [40] can be used to train a large model with various optimizations, it takes excessive extra critical path costs for offloading activations and optimizer states to CPU memory [27] and NVMe storage [41], or sharding them across GPUs [29]. Moreover, its scalability [8] is bounded by communication on low-end networks and the size of a total batch (a set of data for producing each parameter update).

Tensor Parallelism (TP). Tensor (Model) parallelism [25] partitions input and parameter tensors of a layer (e.g., transformer multi-head self-attention layer [30]) across GPUs. Within each repeated (transformer) block (a set of layers), during both forward pass and backward pass, TP requires an all-reduce communication (i.e., TP.sync) to aggregate the tensors between repeated blocks, which typically lies on the critical path and is network-intense. Therefore, TP is usually deployed across GPUs within the same server to use fast intra-server GPU-to-GPU links (e.g., NVLink [42]). TP is mainly adopted as a complementary technique to help existing parallel training systems [8], [19], [25], [41] to support larger transformer layers.

Pipeline Parallelism (PP). Pipeline (model) parallelism [10], [11], [20], [22], [43] shards the layers of a model across multiple GPUs; each shard is called a pipeline stage; activation tensors are propagated between stages via a point-to-point communication (i.e., PP.sync). A total batch is split into micro-batches (a micro-batch is the minimum unit for each GPU's forward and backward pass computations); execution is then pipelined across micro-batches. When used on symmetric models, each stage (GPU) can be assigned an equal number of layers to maximize pipeline efficiency [8].

To retain the convergence guarantee of Stochastic Gradient Descent [44] training, existing PP-enabled systems [19], [22], [45] need to insert a pipeline flush between each two parameter updates, where the systems wait until all current micro-batches in the pipeline finish computing and perform a parameter update, and then restart the upcoming pipelined training iteration. Inserting such a pipeline flush inevitably causes pipeline bubbles (i.e., work idling), as shown in Figure 4. Existing PP scheduling schemes are in the following two categories, in terms of how forward passes are interleaved with backward passes in a pipeline.

AFAB Scheduling. GPipe [22] proposes the all forward all backward (AFAB) scheduling where on each pipeline stage, the forward passes for all micro-batches of a total batch are first executed, followed by backward passes for all micro-batches. For its simplicity and easy-to-integrate nature, AFAB scheduling is widely adopted by systems such as HetPipe [45] and DeepSpeed 3D [19].

1F1B Scheduling. Pipedream [10] proposes the 1F1B scheduling where one backward pass immediately preempts the execution as soon as its required forward pass is finished (for the last stage) or its depending backward passes are finished (for other stages). 1F1B scheduling is adopted by Megatron [8] and recent pure pipeline parallel training systems [22] (e.g., Out-of-Order [21]) with a flush inserted. Compared with AFAB scheduling, 1F1B scheduling costs less GPU memory footprint. Nevertheless, we embrace a CPU offloading scheme (§4) of activation checkpoints to make the AIAO scheduling of FOLD3D not abuse GPU memory.

Overall, despite the differences between the above scheduling algorithms in terms of how forward passes and backward passes are interleaved, all existing pipeline scheduling algorithms are FIFO-based scheduling, as shown in Figure 1b. Specifically, from the view of forward pass computational tasks and backward pass computational tasks separately, micro-batches are executed in the order of they being fed into the execution queue; and later enqueued micro-batches need to wait for the dequeueing of the previous micro-batches' tasks. In this paper, instead of following this FIFO principle, FOLD3D enqueues all the micro-batches to conduct its subtle scheduling (§3.2) of computational tasks that alleviates the serialization problem in existing 3D parallel training systems.

2.2 3D Parallelism

Despite many efforts made towards all the three aforementioned scaling dimensions of parallel training, none of a single scaling dimension could scale infinitely. The reason is that a single scaling dimension may be bounded by various scaling efficiency bounds [8]: TP incurs frequent and high-volume intra-server communication tasks and thus is only suitable within a server (host); DP is bounded by cross-server DP.sync communication tasks and the total batch size [8]; PP is bounded by bubbles and the total number of layers [10]. 3-Dimensional Parallelism (3D) combines all these three dimensions so that when one dimension reaches its scaling efficiency bounds, a 3D parallel system can scale along other dimensions.

The Serialization Problem Existing 3D parallel training systems all suffer from the serialization problem: most of the communicational tasks are serialized after the computational tasks and scattered along the performance critical path. When these systems are deployed on commodity clouds with a few hundred of Gbps networks, the serialization problem is getting much more pronounced. Compared to the reported experiments from Megatron [8] over a cluster of 256 A100 GPUs with 1.6Tbps dedicated inter-host links, the per-GPU hardware utilization sharply dropped from around 140 TFLOPs to 67.4 TFLOPs, when training a GPT3-18B model over a 200Gbps cloud network. Note that in the commodity cloud, each tenant can only get 70 to 80 Gbps bandwidth (§6).

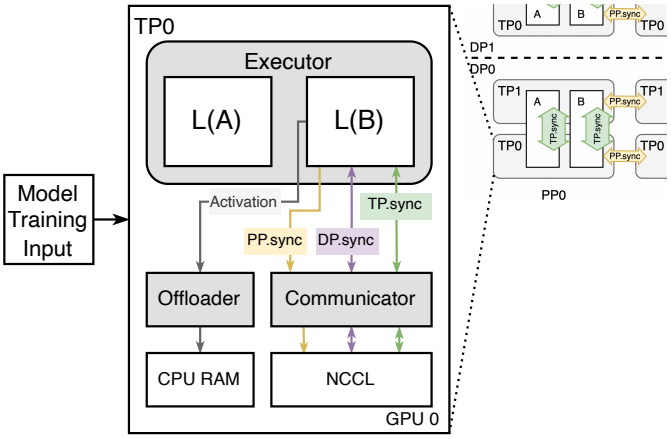


Fig. 3: FOLD3D's Architecture. L(A) and L(B) mean layer A and layer B respectively. The gray boxes are FOLD3D's runtime components.

3 FOLD3D SYSTEM

3.1 Architecture Overview

Figure 3 shows the architecture of FOLD3D. To deploy a model for training, a user needs to feed the model and the 3D parallelism configuration (generated by Megatron-PTD [8] or Piper [16]) into FOLD3D (§4), and FOLD3D will automatically select a proper segment number (§4) and generate an AIAO schedule for the model. Empirically, we find that the 3D parallelism configuration generated by Piper is also optimal for FOLD3D (see §6.2).

For each GPU device, FOLD3D launches one runtime containing an *executor*, a *communicator*, and an *offloader*. On each GPU, FOLD3D's *executor* runs a partition of the AIAO scheduling as shown in Figure 4 and assigns the communication (sync) tasks to FOLD3D's *communicator*. The *communicator* schedules the communication tasks. The *offloader* manages activation checkpoints (§4).

Executor. Each executor is a dedicated main process which manages one GPU device, controls the computation scheduling, and interacts with FOLD3D's communicator and offloader. In particular, the executor runs a static AIAO scheduling, performs the training computation tasks (including both forward pass and backward pass computations), and assigns three types of communication (sync) tasks (including DP.sync, TP.sync, and PP.sync) to the communicator. Meanwhile, the executor informs the offloader of the execution status before each computation task starts, so that the offloader can manage the checkpoint offloading/prefetching without hurting training performance on the critical path.

Communicator. Each communicator is the executor's child thread, which receives communication tasks and schedules the tasks to the underlying communication library (Algorithm 2). We implemented a preemptive communication scheduling mechanism in the communication library. Specifically, the latency-sensitive communication tasks (PP.sync) can preempt the all-reduce communication tasks (DP.sync, TP.sync) to avoid being blocked by the all-reduce tasks (§5).

Offloader. Each offloader is the executor's child thread, which coordinates with the executor and offloads the activation checkpoints to the CPU memory after they are generated in forward passes and prefetches them back to GPU memory before they are required in backward passes.

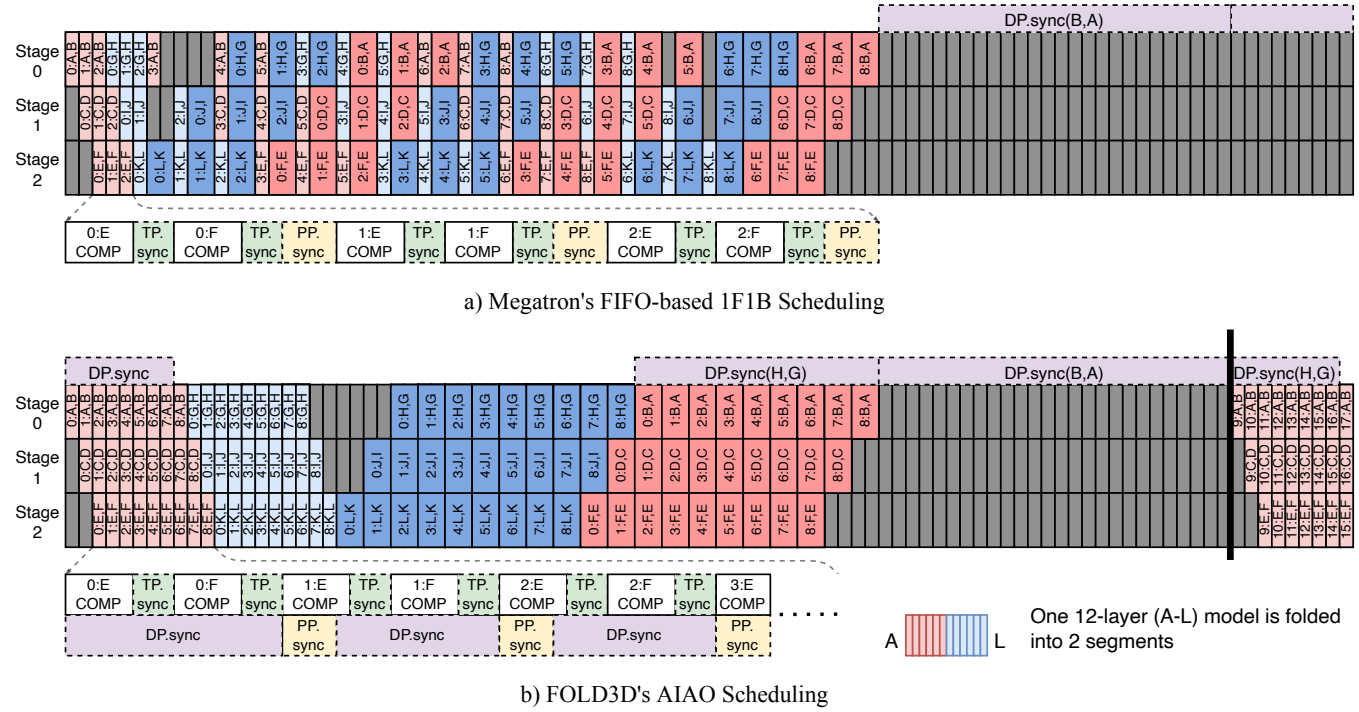
3.2 AIAO Scheduling

We propose AIAO (Figure 4), a new 3D parallel scheduling algorithm that co-schedules and parallelizes the computation and communication tasks to fully (but not overly) utilize both the GPU devices' computation capacity and the networks' communication bandwidth. AIAO works in three steps: first, it folds a model into segments (*step 1*); second, it pipelines each segment across all pipeline stages (*step 2*); third, it schedules the communication tasks to maximally parallelize them with the computation tasks (*step 3*). Same as Megatron, AIAO is bulk synchronous [46], where a pipeline flush is inserted for parameter update to retain the convergence guarantee of Stochastic Gradient Descent [47] training.

Step 1. As shown in Figure 4, the first step is the folding of all layers: given a 3D parallelism configuration with PP stage number denoted as p, each model is divided (folded) into a number (denoted as ns, inferred in §4) of segments, and each segment is further divided into p stages. For example, if one model has 12 layers (from A to L, alphabetically), ns=2, and p=3, FOLD3D assigns GPU 0 with layers $(A, B), (G, H)$; GPU 1 with layers $(C, D), (I, J)$; GPU 2 with layers $(E, F), (K, L)$. Although Megatron [8] already has a segmenting scheme, Megatron's scheme differs from FOLD3D's in purpose: Megatron's scheme is designed to reduce bubbles in its 1F1B pipeline, and the segmenting scheme in FOLD3D's AIAO scheduling is designed to unleash the potential of DP.sync overlapping with computation tasks. Moreover, FOLD3D's segmenting scheme is used to balance the deducted DP.sync tasks (by overlapping) and the increased PP.sync tasks (by folding).

Step 2. The fundamental idea of step 2 is that the pipeline scheduling should be performed in a way that a model layer's gradient should be attained first (thus the layer's computation tasks should be scheduled in a bundle) for decoupling the dependency of this layer's DP.sync communication task with its computation task, so that this DP.sync communication task can be scheduled to overlap other layers' computation tasks. Therefore, in the second step, AIAO schedules the attained computation tasks in a bundled and spiral way, where each segment is further split and pipelined across all stages during the default injection of multiple micro-batches in any PP-enabled training schedules (§2.2). For example, in Figure 4, during the forward passes of AIAO, the first segment is further partitioned into p (three) stages, and the first segment $((A, B), (C, D), (E, F))$ is injected with nine micro-batches (defined by the user) in a pipelined way in their forward passes. The following segments are then executed subsequently.

During AIAO's backward passes, reversely, the last segment is first executed $((L, K), (J, I), (H, G))$ in the pipeline. The reason is that a backward pass must always start from the last layer of a DNN model [48]. Meanwhile, in each stage, after finishing the second segment's backward pass computation tasks (which take roughly twice the time



a) Megatron's FIFO-based 1F1B Scheduling

b) FOLD3D's AIAO Scheduling

Fig. 4: Comparison between Megatron's serialized 1F1B scheduling and FOLD3D's AIAO scheduling.

of their corresponding forward passes due to activation checkpointing to save GPU memory in all PP-enabled training systems' schedules [22]), the DP.sync tasks (e.g., DP.sync(H, G)) of all layers in this segment can immediately be launched, and the first segment's computation tasks can start in parallel. When $ns = 1$, AIAO essentially becomes GPipe's AFAB schedule. When $ns = x$, the DP.sync tasks of $x - 1$ segments can be overlapped with computations, and only the DP.sync of 1 (the first) segment cannot be overlapped with other tasks.

Step 3. An ideal case of overlapping DP.sync tasks with computation is that DP.sync tasks of a segment's layers are faster than another segment's backward pass computation tasks. For instance, the DP.sync(H, G) finishes no later than the backward pass of (B, A). In this case, only the first segment's DP.sync tasks lie on AIAO's performance critical path, because the first segment's next forward pass should wait until these DP.sync tasks finish, which is the optimal case as discussed in §2.2.

However, in a commodity cloud's network, the finish time of a segment's DP.sync tasks (e.g., DP.sync(H, G)) can be longer than the overlapped backward pass (e.g., the backward pass of (B, A)). Therefore, FOLD3D truncates the longer part of the DP.sync tasks and overlaps this part with a corresponding forward pass (e.g., the forward pass of (A, B)) in the next iteration (details are in §4). For instance, in Table 2, for FOLD3D, the time spent in the "**DP.sync**" column of all segments mostly overlapped with the time spent in the "**Bwd**" column of all (other) segments; for Megatron, these two columns were serialized in its training performance critical path.

3.3 Performance Modeling

Critical Path Analysis. In conventional 3D parallel training systems [8], [16], [19], the execution time (i.e., defined as the critical path) of one iteration processing a whole data batch can be divided into computation time T^{comp}, communication time T^{comm}, and bubble time T^{bubble}. Generally, the performance model used for evaluating 3D configurations in Megatron, Alpa and Piper can be unified as:

$$T^{comp} + T^{comm} + T^{bubble} \tag{1}$$

FOLD3D overlaps communication with computation, and the communication time in FOLD3D can be further divided into overlapping time T^{comm}_{ol} and non-overlapping time T^{comm}_{nol}. The critical path of FOLD3D is thus defined as:

$$max(T^{comp}, T^{comm}_{ol}) + T^{comm}_{nol} + T^{bubble} \tag{2}$$

As formulated in recent work [8], [16], the computation time T^{comp} is orthogonal to the scheduling strategy and relates only to the given 3D configuration. Therefore, given the same DNN model and 3D configuration, the T^{comp} of FOLD3D should perform the same as that of Megatron and any other 3D parallel training systems.

T^{bubble} is the bubble time in the pipeline, and is calculated as the sum of startup times of all pipeline stages. The startup time of a pipeline stage is defined as the sum of forward and backward times of its first micro-batch.

We denote the pipeline stage number as p. According to the segment-based scheduling, the bubble in FOLD3D consists of $p - 1$ forward passes and $p - 1$ backward passes of a segment's micro-batch. Given the segment number ns and the micro-batch number ms, T^{bubble} is:

$$(p - 1) * \frac{T^{comp}}{ns * ms} \tag{3}$$

Data parallelism can be performed inside a host and across hosts. Thus, we define intra-host data parallel size d_{intra} as the number of GPUs in the same data parallel group on a host and inter-host data parallel size d_{inter} as

the number of hosts in a data parallel group. Assume w is the total model parameter size that all GPUs in a host contain, the data parallel communication time T_{dp}^{comm} is $\frac{2(d_{inter}-1)w}{d_{inter}r}$, where r is the network bandwidth of a host. We assume the traditional all-reduce [18] communication for the DP.sync here. The data parallel communication of that all ns segments except for the first segment in a stage can overlap with the computation.

In FOLD3D, a tensor is transmitted in both the forward and backward passes of a micro-batch. The tensor size equals the activation size a of a single layer in the model. Assume there are total bs tensors transmitted in a stage, the pipeline parallel communication time T_{pp}^{comm} is $\frac{bs*a}{d_{inter}*r}$. The pipeline parallel communication between stages can be transmitted asynchronously. The pipeline parallel communication of all ms micro-batches except for the first micro-batch of the first segment in a stage can overlap with the computation. The overlapping communication T_{ol}^{comm} is :

$$\frac{ms-1}{ms}T_{pp}^{comm}+\frac{s-1}{s}T_{dp}^{comm}-T^{comp} \tag{4}$$

Thus, the non-overlapping communication time T_{nol}^{comm} is:

$$\frac{T_{pp}^{comm}}{ms}+\frac{T_{dp}^{comm}}{s}+T_{tp}^{comm} \tag{5}$$

Memory Analysis. The major difference between AIAO scheduling and FIFO-based scheduling is that an activation checkpoint on average incurs a longer lifecycle in AIAO, making the peak working-set memory of FOLD3D larger than the other FIFO-based 3D training systems. We take Megatron's interleaved 1F1B scheduling [8], [16] as an example. The peak memory consumption for activation checkpoints of Megatron is:

$$(p*(ns+1)-1)*SizeOf(checkpoints) \tag{6}$$

The peak memory consumption for activation checkpoints of FOLD3D is:

$$ms*ns*SizeOf(checkpoints) \tag{7}$$

Overall, the Megatron's memory consumption for stashing activation checkpoints is only related to the pipeline stage number p and segment number ns, while FOLD3D's total memory consumption for stashing the activation checkpoints is proportional to the number of micro-batches (ms), which makes FOLD3D's memory consumption larger than Megatron's in most cases. Nevertheless, FOLD3D's offloading mechanism shifts this extra memory burden to the CPU memory, making FOLD3D incur negligible extra GPU memory usage (see Table 2).

4 FOLD3D RUNTIME

FOLD3D takes a model's shape setting and training hyperparameter as inputs and trains the model with 3D parallelism on a GPU cluster. The model shape setting includes hidden size, number of attention heads, number of layers, etc. The training hyperparameter contains learning rate, weight decay, etc. The 3D parallelism setting includes pipeline parallelism size p, tensor parallelism size t and data parallelism size d.

FOLD3D automatically determines AIAO's segment number for the given model and parallelism setting in order to reach a high training performance. Specifically, an ideal segment number should balance the DP.sync (network bandwidth-hungry) and PP.sync (latency-sensitive) tasks, and should overlap the communication and computation tasks as much as possible (§3.2), as shown in Figure 4. If the number of segments (ns) is larger, FOLD3D can move more DP.sync tasks off the critical path, and AIAO's pipeline bubble ratio can decrease. However, these benefits do not come for free: increasing the segment number to ns will invoke $ns-1$ more times of PP.sync tasks (Figure 4b). Although FOLD3D overlaps PP.sync tasks with computation tasks through asynchronous transfer (§5), the PP.sync tasks may still block the DP.sync tasks, because both these two communication tasks contend for the same network. Therefore, FOLD3D determines a near-optimal segment number (ns) heuristically. FOLD3D increases ns until the combination of PP.sync and DP.sync tasks exceed the computation time being overlapped.

The executor realizes the AIAO scheduling, given the 3D parallelism strategy (p,t,d) and the segment number ns. Algorithm 1 describes the executor's logic: it invokes all sync tasks based on the current injected micro-batch ID and the current GPU's segment ID to determine its upcoming communication and computation tasks' interleaving. It first executes all micro-batches' (in this training iteration) forward passes (line 7) and then executes all micro-batches' backward passes (line 13). After that, the pipeline flush (line 4) is performed to synchronize all the gradients along the DP dimension and update the model parameters. During the computation tasks, the executor assigns the generated communication tasks to the communicator with each communicated object reference and its current execution status.

Algorithm 1: FOLD3D Executor

```
   Input: Training iteration T; Micro-batch number m;
          Segment number ns;
1  for i = 1 to T do
2      for j = 1 to ns do
3          seg ← getSegment(j);
           // wait for line 13 in DPComm to
              finish
4          seg.flush();
5          for k = 1 to m do
               // prepared by recv() in PPComm
6              input ← seg.getForwardInput(k);
7              output ← seg.runForward(input);
               // invoke send() in PPComm
8              seg.setForwardOutput(k, output);
9      for j = ns to 1 do
10         seg ← getSegment(j);
11         for k = 1 to m do
               // prepared by recv() in PPComm
12             input ← seg.getBackwardInput(k);
13             output ← seg.runBackward(input);
               // invoke send() in PPComm
14             seg.setBackwardOutput(k, output);
           // invoke line 6 in DPComm
15         seg.setBackwardDone();
```

Algorithm 2 describes FOLD3D's communicator logic. It splits the DP.sync task of each segment into two subsets. The first subset is launched after it is generated, and this subset will be overlapped with the upcoming segment's backward pass (line 10). The second subset will be overlapped with the corresponding forward pass computation (line 13). The scheduling outcome is depicted in Figure 4. FOLD3D communicator automatically decides the split ratio of the two subsets based on the runtime-collected computation time of backward pass and forward pass tasks correspondingly.

FOLD3D communicator issues pipeline parallel send() and recv() operations per forward or backward pass (lines 21, 24). At the beginning of each computation task, FOLD3D communicator issues the recv() operation for the next computation task's input tensors. The recv() operation should finish before the next computation task starts. At the end of each computation task, FOLD3D communicator calls send() with the output tensors to transfer them to the next pipeline stage. FOLD3D communicator issues TP.sync tasks during the forward and backward passes. FOLD3D executor waits until the TP.sync tasks finish and then continues the computation.

Algorithm 2: FOLD3D Communicator

Input: Training iteration T; Micro-batch number m; Segment number ns; DP.sync split ratio r

1 **Procedure** `DPComm()`:
2 **for** $i = 1$ **to** T **do**
3 $DP_{fwd} \leftarrow \emptyset$;
4 **for** $j = ns$ **to** 1 **do**
5 $seg \leftarrow getSegment(j)$;
6 $seg.waitBackwardDone()$;
7 $task \leftarrow seg.DPSync()$;
8 $task_{fwd}, task_{bwd} \leftarrow task.split(r)$;
9 $DP_{fwd}.append(task_{fwd})$;
 `// finish backward DP.sync task`
10 $task_{bwd}.launch()$;
11 **for** $j = 1$ **to** ns **do**
12 $task_{fwd} \leftarrow DP_{fwd}.popLast()$;
 `// finish forward DP.sync task`
13 $task_{fwd}.launch()$;
14 **Procedure** `PPComm()`:
15 **for** $i = 1$ **to** T **do**
16 **for** $j = 1$ **to** ns **do** `//` $j = ns$ **to** 1 `for backward`
17 $seg \leftarrow getSegment(j)$;
18 $input \leftarrow recv(1)$;
19 $seg.setInput(1, input)$;
20 **for** $k = 1$ **to** $m - 1$ **do**
21 $input \leftarrow recv(k + 1)$;
22 $seg.setInput(k + 1, input)$;
23 $output \leftarrow seg.getOutput(k)$;
24 $send(k, output)$;
25 $output \leftarrow seg.getOutput(m)$;
26 $send(m, output)$;

Intra-Segment Offloading. The offloader incorporates both activation checkpointing and cpu offloading. Activation checkpointing [49] is essential for greatly reducing GPU memory footprint (i.e., the activation tensors) in existing PP-enabled training systems by paying extra re-computation time of GPU ALUs. Activation checkpointing only stashes output activations (i.e., checkpoints) of selective layers, and the rest activations are recomputed in the backward pass by running the forward pass again.

The offloader decides on a proper set of activation checkpoints, which does not defer the progress of backward pass, but achieves the minimum peak memory footprint [49]. Compared to Megatron's 1F1B scheduling, FOLD3D's AIAO scheduling inevitably incurs larger GPU memory (§2.2). This is because AIAO requires all the forward passes of all segments to be finished before any backward pass starts (see Figure 4). The activation size may still exceed the GPU memory even with activation checkpointing enabled in FOLD3D. Therefore, the offloader offloads activation checkpoints to CPU memory, a common trick adopted from existing systems (e.g., DeepSpeed [19]). All checkpoints generated by a micro-batch are offloaded during the next micro-batch's forward pass. Then the offloader pre-fetches the checkpoints of each micro-batch from CPU memory during the previous micro-batch's backward pass. Evaluation shows that FOLD3D's pre-fecthings/offloadings were overlapped by computation tasks and caused negligible training slowdown (Table 4).

Inter-Segment Lazy Communication. For all micro-batches of segment i (e.g., segment 0 in red in Figure 4b), the last pipeline stage of this segment (stage 2) has to send the output tensors (of layer F) to segment $i + 1$'s first stage (stage 0). However, the execution of segment $i+1$ on stage 0 will not start until segment i finishes. If using a naive GPU-to-GPU direct communication, the stashed output tensors will cause extra GPU memory consumption (e.g., the output tensors of micro-batch 0-8 will be stashed in stage 0's GPUs in Figure 4b). We shift this part of extra GPU memory to CPU hosts by FOLD3D's inter-segment lazy communication mechanism, in which the output tensors from the last stage's GPUs (e.g., stage 2's GPUs in Figure 4b) will be directly sent to the remote CPU memory of the first pipeline stage (stage 0), and these tensors on the remote CPU memory will be lazily loaded into GPU memory when the tensors are used in related computational tasks.

Overall, these runtime algorithms do not affect the bulk synchronous training convergence for three reasons. First, each segment collects the gradients along the DP dimension through DP.sync tasks (see lines 10, 13 in Algorithm 2) and each segment has gradients with respect to all samples in an iteration. Second, FOLD3D ensures that the gradients of each segment are updated to the model parameters before the next iteration of this segment begins (see line 4 in Algorithm 1). Third, FOLD3D does not alter tensors transmitted between GPUs or tensors fed into the model.

5 IMPLEMENTATION

5.1 Preemptive communication scheduling

In FOLD3D's scheduling, DP.sync tasks are not only overlapped with computation but are also overlapped with PP.sync tasks. Concurrent DP.sync and PP.sync tasks may contend for network bandwidth, and both tasks slow down. Although most of the DP.sync and PP.sync tasks are overlapped with computation in FOLD3D, the PP.sync tasks dur-

ing the pipeline warmup period still stay in the performance critical path.

FOLD3D incorporates preemptive communication scheduling to ensure that the PP.sync time does not increase when overlapping with DP.sync tasks. Specifically, when a PP.sync task arrives, FOLD3D pauses the DP.sync tasks in the same node. For a PP.sync send task, only the DP.sync tasks currently sending data are stopped. For a PP.sync receive task, the DP.sync tasks only stop receiving data when the PP.sync task starts to receive data. When a DP.sync task stops receiving data, it first saves the data already received in its buffer, and then sends an interruption signal to the corresponding sender.

5.2 CPU Offloading

FOLD3D incorporates CPU offloading to mitigate the increased GPU memory burden caused by FOLD3D's scheduling. The activation tensors are continuously moved from GPU memory to CPU memory during the forward pass and moved back to GPU memory in the backward pass. When transferring data from the CPU memory to GPU, GPU requires the CPU memory page to be *pinned* (page-locked). Otherwise, a temporary pinned page is created and data is first copied to the pinned page and then transferred to the GPU. This is because the OS would swap an unpinned page to the disk if the page is inactive. For efficient data transfer, FOLD3D preallocates a pinned CPU memory buffer at the start of training to store the activation tensors. The buffer size is determined by profiling the total activation tensor size in an iteration.

We assign an individual CUDA stream for CPU offloading so that the CPU offloading does block the computation and communication tasks. In the backward pass, the activations are moved back to the GPU according to the order they are used. FOLD3D synchronizes the computation stream with the offloading stream before each activation is used for recomputation to ensure data correctness.

When sequence parallelism is enabled alongside tensor parallelism, the activations are partitioned across the tensor parallel ranks. As a result, each GPU only offloads its own activation partitions to the CPU memory.

The kernel launch overhead becomes significant when we invoke a GPU kernel for each activation tensor to be transferred. FOLD3D introduces a technique named *batched CPU offloading* to reduce the kernel launch overhead. Specifically, in the forward pass, a layer's output tensor saved as the checkpoint is not moved out of GPU memory immediately after it is used by the next layer. Instead, we batch multiple activation tensors together and transfer them between GPU memory and CPU memory in a single kernel. Tensors are also moved back to GPU in batches. By doing so, we can achieve higher PCIe utilization and reduce the CPU offloading time.

6 EVALUATION

Testbeds. We performed the experiments on two clusters. The first cluster is a public commodity cloud consisting of 8 nodes with in total 64 NVIDIA V100 GPUs. Each node is an AWS EC2 p3dn.24xlarge instance which has 96 vCPUs, 1.2TB memory and 8 Nvidia Tesla V100 GPUs (each has 32GB memory and 125 FP16 TFLOPs). GPUs in a node are connected by NVLink, and nodes are connected over a 100Gbps network. The second cluster is a private laboratory cloud containing 32 nodes with a total of 256 NVIDIA A100 GPUs. Each node has 128 Intel 6248R CPUs, 2.0TB memory and 8 Nvidia A100 GPUs (each has 40GB memory and 312 FP16 TFLOPs). GPUs in a node are connected by NVLink, and nodes are connected over 200Gbps Infiniband. Unless otherwise specified, we used 16 nodes with a total of 128 A100 GPUs as our default testbed.

Baselines. We took Megatron v3.0 (Megatron-SP) [28], Megatron v2.5 (Megatron-PTD) [8], DeepSpeed 3D (DSpeed3D) [19], and DeepSpeed ZeRO3 (DSpeedZ3) [29] as our baselines. Megatron-SP is the latest 3D parallel training system that was reported to achieve almost linear scaling efficiency. Megatron-PTD is the system used in Megatron's earlier paper [8]. DSpeedZ3 is a powerful data parallel training system that incorporates a set of memory optimization techniques. Microsoft's DSpeed3D is a well-engineered system which extends data parallelism optimized by DeepSpeed ZeRO with tensor parallelism [25] and pipeline parallelism [10], [24] to break the scaling efficiency bounds of data parallelism. We ran these two DeepSpeed systems in DeepSpeed v0.5.5 environment. Sequence parallelism [28] was integrated into Megatron, DSpeed3D and FOLD3D to reduce the activation size and support larger models.

Baseline settings. We used two 3D parallel configurations for the experiments. The first configuration was chosen following the instructions provided in Megatron-PTD. Specifically, given a DNN model, we first scaled along the tensor parallel dimension within hosts and then the pipeline parallel dimension until the model's parameters and activations can be fit into GPU memory. Then, we scaled along the data parallel dimension to use up all GPUs. The second configuration was selected by Piper, which proposed an efficient optimization algorithm to find the best 3D parallel configuration for its corresponding 3D parallel performance modeling. For both Megatron-PTD and Megatron-SP, we adopted the interleaved schedule introduced in its paper to reduce the pipeline bubble. The best interleaved schedule was selected by trials and chosen with the highest throughput produced, as no determined selection instruction is provided by Megatron. Megatron-PTD/Megatron-SP can overlap most of the PP.sync tasks with computation when enabling its interleaved schedule, but the PP.sync tasks during the pipeline warmup period have to be in the performance critical path. DSpeed3D has to left all the PP.sync tasks in the performance critical path since it adopts the 1F1B scheduling. This is because when overlapping PP.sync tasks with computation in 1F1B scheduling, two simultaneous send/recv operations between a pair of GPUs may potentially cause deadlock [50].

Models and Datasets. We evaluated five giant transformer models which cover all the large transformer models evaluated by recent large model training systems [8]. Specifically, we covered major pretraining transformer models (GPT [1], BERT [31], CPM [32], Turing-NLG [29] and T5 [33]) and their respective datasets. GPT and Turing-NLG use OpenWebText [51] dataset, BERT uses Wikipedia [52] dataset, CPM uses WuDao Corpus [53] dataset, and T5 uses

Model	#Layers	Hid. Size	#Heads
GPT-3 14B	32	6144	48
GPT-3 18B	40	6144	48
GPT-3 39B	48	8192	64
GPT-3 81B	64	10240	80
BERT	72	7344	48
T-NLG	80	4256	28
CPM	48	5120	64
T5	24	1024	128

TABLE 1: Different models we used during the evaluation. Hid. Size stands for the hidden size of a layer.

c4/realnewslike [54] dataset.

Metrics. We measured FOLD3D's training performance by per-GPU throughput. The throughput is calculated by the TFLOPs metric, whose approximation formulas (for transformer blocks) are from Megatron [8] for fair comparisons, and further specifications can be found in their paper.

Model Configurations. Table 1 shows all model settings used in this paper. Each model's configuration is the same as the official specifications or settings evaluated by previous works. Moreover, to better understand FOLD3D, we evaluated GPT-3 models with various model shapes and parameter sizes. We will specify how these settings are selected when they are used. Without further specifications, the micro-batch size used in our experiments was 4, which was large enough to saturate a GPU's computation while leaving enough GPU memory space for memory footprints during training.

We focus on four questions. §6.1: How does FOLD3D perform compared to the baselines? §6.2: How does FOLD3D perform with different parallel configurations? §6.3: How robust is FOLD3D's high performance under different batch sizes and network bandwidths? §6.4: How effective are FOLD3D and its components?

6.1 End-to-End Performance

Table 2 shows five training systems' per-GPU throughput when training GPT-3 models on 64 V100 GPUs and 128 A100 GPUs. We present the detailed settings (e.g., model, batch size and parallel configuration), as well as various breakdown results, in Table 2. Column "**ExCMem.**" stands for the extra CPU memory brought by the novel AIAO scheduling of FOLD3D. The extra CPU memory is the peak CPU memory occupied by the offloaded checkpoint activations in a host, and excludes the CPU memory used by Python program and PyTorch library. Column "**ExCMem.**" is not applicable to Megatron and DSpeed3D since these systems only store the activation tensors in GPU memory. Column "**ExCMem.**" is not applicable to the pure data parallel training system DSpeedZ3 because Column "**ExCMem.**" evaluates the CPU memory usage caused by pipeline parallelism. Besides the extra CPU memory, we have also evaluated the total CPU memory usage of all systems, and we report the results in §6.1. Columns "**Bubble**" and "**PP.sync**" are not applicable to DSpeedZ3 because DSpeedZ3 is a pure data parallel training system.

All models' shape configurations used in this evaluation followed the models' original papers. To compare with Megatron-PTD, we evaluated GPT-3 18B, which is the model Megatron-PTD used in its paper on 256 GPUs. We then used GPT-3 39B to evaluate FOLD3D's performance on larger models, and compared the results with Megatron-SP. We selected the batch size that achieved the shortest training time making the model converged. Details of the batch size selection are elaborated in §6.3.

Overall, FOLD3D achieved the highest throughput (116.1 TFLOPS and 51.1 TFLOPS) on both A100 and V100 clusters; to the best of our knowledge, this throughput is higher than the highest per-GPU throughput publicly reported on 64 V100 GPUs with similar models. Specifically, DeepSpeed reported a publicly highest per-GPU throughput of 41.4 TFLOPS [29] on training the model of the same size, but with a much faster cross-server link of 800Gbps (Nvidia DGX-2) than the 100Gbps network in our evaluation. FOLD3D achieved 31.5%-42.1% speedup over Megatron-SP on 128 A100 GPUs and 25.2%-33.0% over Megatron-SP on 64 V100 GPUs.

Table 2 reveals FOLD3D's high performance came from both the reduced (overlapped) DP.sync and PP.sync communications on the performance critical path. We observed that the network was saturated by "**DP.sync**" for all five training systems. On the training performance critical path of both Megatron and FOLD3D, both their throughput mainly depends on the sum of "**Fwd.**", "**Bwd.**", "**DP.sync**", and "**PP.sync**". However, FOLD3D's "**Fwd.**" and "**Bwd.**" are overlapped with most of the DP.sync and PP.sync tasks (see Figure 4b). For example, on the V100 cluster, FOLD3D reduced the DP.sync time on the performance critical (non-overlapped) path from 2.50s to 0.67s and the PP.sync time on the performance critical path from 0.95s to 0.41s, respectively.

FOLD3D outperformed the baselines under both the parallel configurations derived by Megatron-PTD and Piper. This is because that DP.sync took a large portion of the iteration time for these parallel configurations. Megatron-PTD increases PP and TP sizes until the model split can be fit into GPU memory, and then enlarges DP to use all GPUs. In such a case, the PP size and TP size are minimized while the DP size is maximized. Meanwhile, Megatron-PTD used most of the GPU memory to accommodate the model parameters and their corresponding gradients. This leads to the extremely large gradients to be synchronized in each GPU. The large per-GPU gradient volume and the large DP size lead to the substantial DP.sync of Megatron-PTD.

Even though Piper automatically finds the best parallel configuration that maximizes the training throughput, DP.sync still accounted for 28.6% of the training time in our evaluation. This is because the decrease of DP.sync time always comes with the increase of PP.sync time and pipeline bubble time, which inevitably increases the overall iteration time. In 3D parallel training, the way to mitigate the DP.sync time is to increase the PP size. By doing so, both the DP size and the gradients needed to be synchronized per GPU decrease. However, both the PP.sync time and the pipeline bubble increase as well.

We have collected both GPU and CPU memory usages during evaluation. The memory usage (i.e., the sum of GPU memory and CPU memory usages) of FOLD3D is larger than the baselines, and the extra memory overhead comes from the novel AIAO scheduling of FOLD3D. The AIAO scheduling requires FOLD3D to store the checkpoint activations generated during the forward pass of all the micro-batches. Since the checkpoints are further offloaded

GPUs	Model	Batch	System	Conf	(3D), (Seg.)	Fwd.	Bwd.	Bubble	DP.sync	PP.sync	GMem.	ExCMem.	Thrp.	Util%
A100 × 128	GPT-3 18B	256	FOLD3D	PTD-P	(8, 2, 8), (4)	634.0	1549.0	79.4	629.1	193.2	27.3	8.3	95.8	32.0%
			Megatron-PTD		(8, 2, 8)	610.1	1512.4	143.6	2020.0	298.0	26.5	n/a	67.4	21.6%
			DSpeed3D		(8, 2, 8)	624.9	1559.0	290.1	2034.1	385.7	26.0	n/a	61.9	19.8%
			DSpeedZ3	n/a	(128, n/a, 1)	3211.8	3790.3	n/a	2365.2	n/a	10.4	n/a	33.0	10.6%
	GPT-3 39B	256	FOLD3D	Piper	(4, 4, 8), (4)	1239.7	2984.3	213.5	541.9	382.2	30.7	12.9	116.1	37.2%
			Megatron-SP		(4, 4, 8)	1152.0	2825.9	439.0	1976.8	732.5	29.8	n/a	88.3	28.3%
			DSpeed3D		(4, 4, 8)	1186.9	2862.8	803.7	1962.3	1263.8	29.0	n/a	78.3	25.1%
			DSpeedZ3	n/a	(128, n/a, 1)	6794.2	6820.7	n/a	4387.7	n/a	16.5	n/a	36.1	11.6%
V100 × 64	GPT-3 18B	128	FOLD3D	PTD-P	(4, 2, 8), (4)	1540.0	4102.0	181.6	873.1	108.9	27.1	8.1	43.5	34.8%
			Megatron-PTD		(4, 2, 8)	1475.1	3919.5	318.2	3235.6	307.8	26.3	n/a	32.7	26.2%
			DSpeed3D		(4, 2, 8)	1560.1	3955.6	693.4	3211.0	503.0	25.8	n/a	30.2	24.2%
			DSpeedZ3	n/a	(64, n/a, 1)	5173.6	5420.1	n/a	3786.1	n/a	10.1	n/a	21.6	17.3%
	GPT-3 39B	128	FOLD3D	Piper	(2, 4, 8), (4)	2920.6	7928.3	508.5	670.2	413.2	30.5	12.9	51.6	41.3%
			Megatron-SP		(2, 4, 8)	2865.3	7836.9	1159.7	2505.0	951.7	29.4	n/a	41.2	32.9%
			DSpeed3D		(2, 4, 8)	2828.3	7852.8	2012.5	2456.5	1681.1	28.7	n/a	38.3	30.6%
			DSpeedZ3	n/a	(64, n/a, 1)	9517.7	9913.8	n/a	7142.3	n/a	16.3	n/a	24.7	19.8%

TABLE 2: Breakdown of performance critical path for each system training GPT-3. Tuples in **(3D), (Seg.)** column stands for (DP, PP, TP), (segment numbers); **Fwd.** stands for forward computing time, **Bwd.** stands for backward computing time. **Thrp.** stands for per-GPU throughput in TFLOPs and **Util%** stands for ratio of the measured throughput to the theoretical peak throughput provided by Nvidia. **GMem.** stands for the peak GPU memory usage (in GB) of a GPU. **ExCMem.** stands for the extra CPU memory brought by the novel AIAO scheduling of FOLD3D and represents the peak CPU memory (in GB) occupied by the offloaded checkpoint activations in a host. Both **PP.sync** and **DP.sync** contain only non-overlapped communication time. **n/a** means the column is not applicable to the system as explained in S6.1.

GPUs	Model	Batch	System	Conf	(3D), (Seg.)	Fwd.	Bwd.	Bubble	DP.sync	PP.sync	GMem.	ExCMem.	Thrp.	Util%
A100 × 128	BERT	256	FOLD3D	Piper	(4, 4, 8), (4)	1987.8	5327.0	413.7	670.5	329.8	39.1	17.3	98.1	31.4%
			Megatron-SP		(4, 4, 8)	1849.4	5242.1	809.8	2284.8	572.5	38.4	n/a	77.8	24.9%
			DSpeed3D		(4, 4, 8)	1870.7	5378.1	1526.8	2356.3	976.4	37.5	n/a	71.2	22.8%
			DSpeedZ3	n/a	(128, n/a, 1)	7615.7	7746.2	n/a	5126.5	n/a	11.3	n/a	51.0	16.3%
	T-NLG	256	FOLD3D	Piper	(8, 2, 8), (4)	1221.0	3574.7	162.7	545.9	227.9	25.7	11.2	96.9	30.4%
			Megatron-SP		(8, 2, 8)	1225.4	3481.3	318.5	1995.3	452.0	24.9	n/a	72.9	23.4%
			DSpeed3D		(8, 2, 8)	1203.1	3489.7	616.6	2054.8	640.4	23.9	n/a	68.4	21.9%
			DSpeedZ3	n/a	(128, n/a, 1)	3783.8	3817.2	n/a	2315.1	n/a	8.5	n/a	59.1	18.9%
	CPM	256	FOLD3D	Piper	(4, 4, 8), (4)	776.7	2021.5	147.7	291.8	161.4	24.5	8.1	91.0	29.1%
			Megatron-SP		(4, 4, 8)	723.4	1986.1	261.8	1084.2	280.9	23.8	n/a	70.9	22.7%
			DSpeed3D		(4, 4, 8)	736.1	1967.4	538.7	1056.3	504.6	22.4	n/a	64.1	20.5%
			DSpeedZ3	n/a	(128, n/a, 1)	4789.2	4543.1	n/a	2589.0	n/a	6.4	n/a	45.9	14.7%
	T5	256	FOLD3D	Piper	(16, 2, 4), (3)	1735.2	4686.0	552.1	867.6	343.1	35.8	14.2	93.7	30.0%
			Megatron-SP		(16, 2, 4)	1649.1	4555.3	898.2	2270.2	768.1	34.7	n/a	74.9	24.0%
			DSpeed3D		(16, 2, 4)	1757.0	4540.6	1648.2	2246.0	1142.6	33.7	n/a	69.8	22.4%
			DSpeedZ3	n/a	(128, n/a, 1)	4716.1	4653.8	n/a	3068.3	n/a	14.1	n/a	64.2	20.6%

TABLE 3: Breakdown of performance critical path for each system training five models. Column name meanings are the same as Table 2. **n/a** means the column is not applicable to the system.

to CPU memory by FOLD3D's offloader, the extra memory used by FOLD3D resides in CPU memory instead of GPU memory. We evaluated extra CPU memory, which is defined as the CPU memory occupied by offloaded checkpoint activations, specifically for FOLD3D. Table 2 shows the GPU memory usages of all systems and the extra CPU memory usage of FOLD3D. The GPU memory used by FOLD3D is comparable to the GPU memory used by Megatron-SP, Megatron-PTD and DSpeed3D. The GPU memory usage of FOLD3D validates the effectiveness of FOLD3D's offloader. When excluding the extra CPU memory used by FOLD3D's offloader, the CPU memory for Python training scripts, dataset loaders, and PyTorch runtime used by FOLD3D also equals that used by Megatron-SP, Megatron-PTD and DSpeed3D. For instance, for the setting that trains a GPT-3 39B model on 128 A100 GPUs in Table 2, when excluding the 12.9GB CPU memory used by FOLD3D's offloader, the remaining CPU memory usage of FOLD3D is 203.4GB, and the total CPU memory usages of Megatron-SP, Megatron-PTD and DSpeed3D are 202.9GB, 200.2GB and 201.7GB.

DSpeedZ3 consumed the smallest GPU memory because it partitions the model parameters, gradients and optimizer states across the data parallel GPUs. Gradient partition combined with parameter partition also makes DSpeedZ3 able to use a scatter-reduce operation for gradient synchronization instead of an all-reduce operation. The DP.sync time is reduced by half compared to the data parallel approach using all-reduce for gradient synchronization. The drawback of DSpeedZ3 is that each GPU has to collect parameters from the other GPUs during both the forward and backward passes. Although DSpeedZ3 overlaps the parameter collection with computation, the parameter collection time is larger than the computation time and dominates the forward and backward passes in our evaluation.

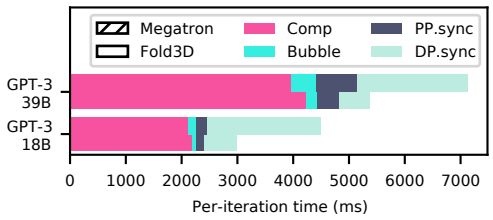


Fig. 5: Breakdown comparison between Megatron-PTD and FOLD3D. The iteration time reduced by FOLD3D matches the performance modeling in §3.3.

We further draw the breakdown results of FOLD3D and Megatron-SP for GPT-3 18B and GPT-3 39B in Figure 5. The results generally matched our performance modeling in § 3.3. For the two models, the DP.sync time of FOLD3D was reduced by 68.9% and 72.6% compared to Megatron-SP. In our performance modeling, the DP.sync time will be reduced by 75% when the segment number is 4. We attribute this discrepancy to the fact that not all GPUs start the DP.sync tasks at exactly the same time. The computation time of FOLD3D is also slightly larger than Megatron-SP for both models. We attribute the slowdown to the overlapping of DP.sync task with computation. As revealed by a recent study [55], when overlapping the all-reduce operation in DP.sync with DNN computation, the all-reduce operation contends for GPU resources with DNN computation. However, these facts only cause the real iteration time less than 5% larger than the performance modeling in our evaluation, and performance modeling is useful to estimate the real performance of FOLD3D.

In Table 3, we show four systems' per-GPU throughput for another four models. We only demonstrate the results for each model under the Piper setting. FOLD3D achieved 1.25x to 1.33x speedup over Megatron-SP, which further confirms that FOLD3D's gain holds for different models. The four models only differ from GPT-3 models in the pre-process and post-process layers. Same as GPT-3 models, majority of the four models are composed of Transform blocks. The models mainly differ in the hidden sizes of the transformer blocks. The hidden size determines the ratio between computation, DP.sync and PP.sync. We further show the results for each model on 64 V100 GPUs in Figure 6.

We conducted a weak scaling study to evaluate FOLD3D's performance on large-scale clusters. In particular, following the common practice of baselines [8], weak scaling is to test a system's throughput on scaling to train a larger model with more GPUs. Figure 7 shows that FOLD3D's throughput was consistently ∼31% higher than both Megatron-SP and DSpeedZ3. FOLD3D still consistently outperformed baselines in all the scales we evaluated. The reason is when the model size and the number of GPUs grow, both computation and communication time increase accordingly. We believe FOLD3D will be able to overlap most of the communication tasks with the computation tasks even in larger scales, e.g., 512 or thousands of GPUs; in contrast, baselines again left all tasks being serialized on the performance critical path.

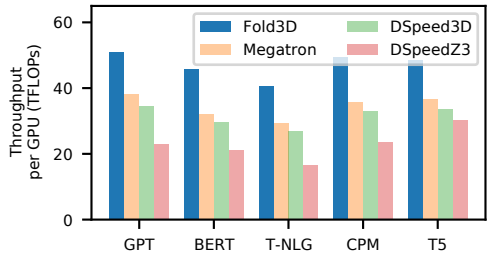


Fig. 6: Per-GPU throughput of different models on 64 V100 GPUs.

In sum, through the end-to-end experiments, FOLD3D was both high-performance and scalable. FOLD3D was reported with speedups over baselines on various commodity cloud settings and various parallel configurations.

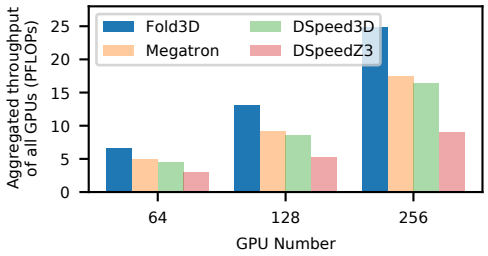


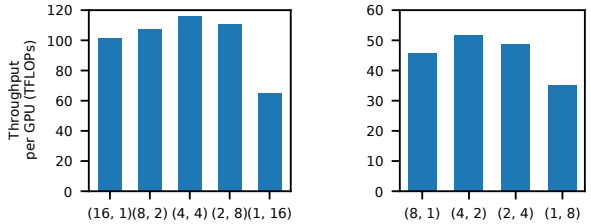
Fig. 7: Weak Scaling of FOLD3D on different amounts of GPUs. FOLD3D consistently achieves higher TFLOPs per GPU under various GPU numbers. We used GPT-3 18B, GP-3 39B, GPT-3 81B for each amount of GPUs.

6.2 Evaluation of Parallel Configurations

In this section, we evaluated the performance of FOLD3D on different parallel configurations. In particular, we fixed the size of one parallel dimension and changed the combination of the other two. We conducted the experiments for model GPT-3 39B on both 64 V100 GPUs and 128 A100 GPUs. For each cluster, the batch size used is the same as the one in Table 2.

6.2.1 PP v.s. DP

In Figure 8, we evaluated the impact of PP and DP sizes on FOLD3D's performance. We set the TP size to 8 (the number of GPUs in a host), which is a common setting for large model training (see Table 3). The Megatron paper's lesson on these two degrees of parallelism is that DP should always be more favorable than PP on their 1.6 Tbps network. However, we found DP and PP should be balanced to reach the peak throughput in our evaluation. This is because a large PP size will incur longer PP.sync time and longer bubble time, while a large DP size will increase the DP.sync time (even when most of the DP.sync is overlapped with computation in FOLD3D).



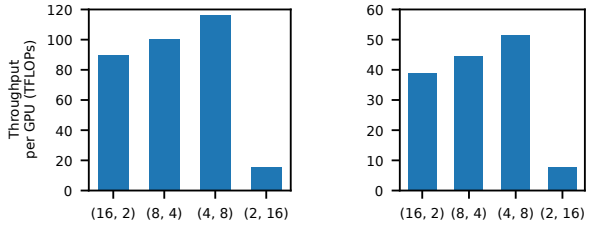
(a) 128 A100 GPUs (b) 64 V100 GPUs

Fig. 8: Throughput per GPU under different (PP size, DP size) combinations. DP and PP should be balanced to reach the peak throughput.

6.2.2 PP v.s. TP

In Figure 9, we evaluated the impact of PP and TP sizes on FOLD3D's performance. We set the DP size to 2 for 64 V100 GPUs and to 4 for 128 A100 GPUs. Overall, our evaluation shows that within a host (TP size less than or equal to 8), on both the V100 GPU cluster and the A100 GPU cluster, TP is more preferred than PP. This is because within a host,

the GPU-to-GPU links are fast enough so that the scaling efficiency of TP (mainly bounded by TP.sync communications) can surpass the efficiency of PP (mainly bounded by flush bubbles). TP size greater than 8 means cross-server TP (because there are 8 GPUs in a node), which is much slower than TP within a server. The same conclusion is reported in Megatron [8].

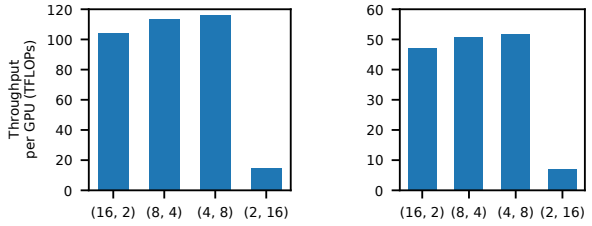


(a) 128 A100 GPUs (b) 64 V100 GPUs

Fig. 9: Throughput per GPU of FOLD3D under different (PP size, TP size) combinations. TP is more preferred than PP within a host.

6.2.3 DP v.s. TP

In Figure 10, we evaluated the impact of DP and TP sizes on the training throughput of FOLD3D. We set the PP size to 4. The figure shows that within both V100 and A100 clusters, TP is more preferred than DP. This is because the TP.sync time was faster than the DP.sync time. When a model's size increases (e.g., increasing the number of DNN layers), the DP.sync time increases faster than the TP.sync time, and TP would still be more preferred.



(a) 128 A100 GPUs (b) 64 V100 GPUs

Fig. 10: Throughput per GPU under different (DP size, TP size) combinations. TP is more preferred than DP within a host.

6.2.4 Impact of Segments

We evaluated the impact of segment number selection on FOLD3D's performance for models GPT-3 39B and GPT-3 14B. For both models, we used the 3D parallel configurations derived by Piper. Since DP and PP should be both preferred and balanced on commodity cloud networks, selecting a proper segment number in FOLD3D is crucial. Although an extremely large segment number will bring a larger overlapping ratio of DP.sync communication tasks with computation tasks (§4), it will also increase the PP.sync costs, as each segment needs to be pipelined across all pipeline stages. We found that in most of our experiments, the best segment number was 2 to 4 for various models, which matched the conclusions we drew from §4, as segment number in this range retains speedup from a highly overlapped portion of communication tasks (50-75%) without incurring too much PP.sync cost.

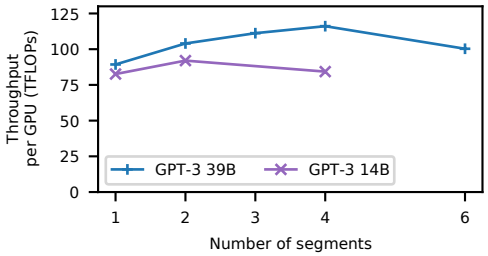
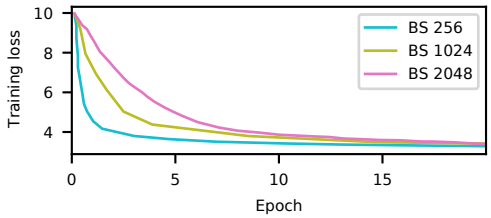
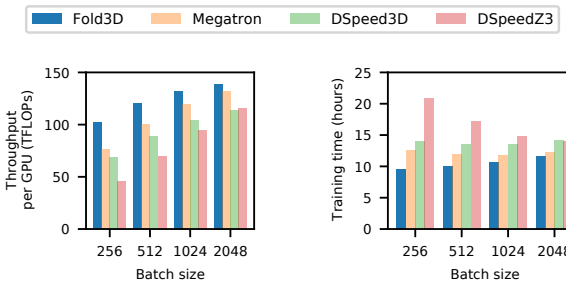


Fig. 11: How the number of segments affects the final throughput. Given a model and its 3D parallelization configuration, FOLD3D's runtime is able to find the optimal segment.

6.3 Ablation Study



(a) BS means batch size. Model converges slower when using larger batch sizes.



(b) FOLD3D achieved larger improvement over baselines under smaller batch size.

(c) 3D parallel training systems had the shortest training time under the smallest batch size.

Fig. 12: (a) Training loss curves under different batch sizes. (b) Throughput per GPU under different batch sizes. (c) Training time required for the training loss to reach 3.3 (the minimum training loss achieved by the model).

The selection of batch size when training large models involves the trade-off between system training throughput and convergence efficiency [56]. When enlarging the batch size, GPUs can achieve higher ALU utilization, but the convergence efficiency becomes lower due to the decrease of gradient noise scale [57]. To demonstrate the relationship between the convergence efficiency and batch size, we trained GPT-3 39B model under different batch sizes. For each batch size we evaluated, we selected the best learning rate and other hyperparameters following approaches from existing works [1]. Figure 12a plots the training loss curves under different batch sizes. When increasing the batch size, the model has to be trained with more epochs although the training throughout increases. Thus, the higher throughput brought by a larger batch size does not necessarily shorten training time. The result of the relationship between

convergence efficiency and batch size also matches recent study [58].

We first evaluated the performance of FOLD3D and baselines under different batch sizes. The result is shown in Figure 12b. When increasing the batch size from 256 to 1024, FOLD3D's throughput improvement over Megatron decreased from 31.5% to 10.7%, and improvement over DSpeedZ3 decreased from 48.2% to 27.2%. This is because the computation time and the overall iteration time increase with the batch size, while the DP.sync time is orthogonal to the batch size and stays roughly the same across various batch sizes. The ratio of the DP.sync time thus decreased and so did FOLD3D's improvement. Although the improvement of FOLD3D over the baselines decreased when enlarging the batch size, we found that the relatively smaller batch size (256) achieved the shortest training time for the model to attain the desired training loss even for Megatron and DSpeedZ3. Figure 12c shows the total training time used for the training loss to achieve 3.3 (the minimum training loss that can be achieved by the given GPT-3 model) under each batch size for all systems.

Our evaluation on AWS cloud shows that the network bandwidth for a single node ranges from 70 to 80 Gbps. We thus evaluated the throughput of FOLD3D and baselines under 25, 40, 100 and 200 Gbps networks on 128 A100 GPUs. Similar to the approach stated above, we chose the best batch size for these systems under each network bandwidth. The best batch sizes for 25, 40, 100 and 200 Gbps networks are 1024, 1024, 512 and 256 respectively. With the decrease of bandwidth, batch size has to be enlarged to achieve higher system throughput and shorten the overall training time. The result is shown in Figure 13. FOLD3D outperformed Megatron-SP by 30.4% to 36.7%, and outperformed DSpeed3D by 38.2% to 45.6%. This is because the ratio of DP.sync to the computation time stayed in the range between 22.3% to 29.6% under all the bandwidths we evaluated. The bandwidth decrement came with batch size enlargement, and thus led to both longer computation time and DP.sync time.

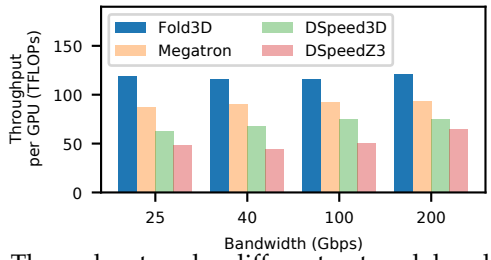


Fig. 13: Throughput under different network bandwidths.

6.4 Effectiveness Analysis

Activation Checkpointing. The activation checkpointing technique [49] in both FOLD3D and Megatron was necessary for conducting all our experiments, because all the baseline systems were with activation checkpointing to support training of large models. We tried to disable activation checkpoiting in our experiments, and all default configurations went to out-of-memory exceptions, indicating that activation checkpointing was necessary.

Checkpoint CPU Offloading. The data transfer rate between the accelerator and the CPU memory, which is bounded by the PCIe bandwidth, always grows proportionally to the accelerator's throughput. In our test environments, the total GPU-to-CPU data transfer rate is 29.6GB/s for 8 A100 GPUs and 13.6GB/s for 8 V100 GPUs, while A100 GPU's peak throughput is 312TFLOPs and V100 GPU's peak throughput is 125TFLOPs.

Model	Mi. batch	Hid	Acti. Size	Fwd.	Bwd.	O/F.
BERT	4	7344	56.6M	4081.2	11271.7	2343.8
GPT-3 39B	4	8192	64.0M	2901.7	7902.3	1472.5
CPM	4	5120	40.0M	1673.2	4707.9	1097.3

TABLE 4: Cost of CPU activation offloading in different models and runtime configurations. Mi. batch stands for micro-batch size and O/F. stands for time cost of offloading opeartions.

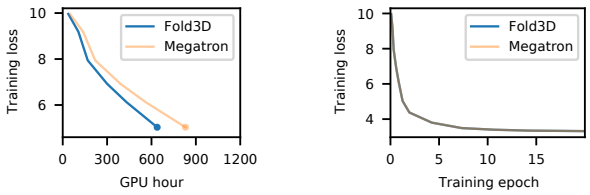
We evaluated the effectiveness of our CPU Offloading of activation checkpoints (§4). Table 4 shows the microevents when training three models on the V100 setting. In particular, given the micro-batch size and hidden size, we collected the activation checkpoint offloading/prefetching time and collected the forward and backward pass time between two activation checkpoints. Overall, the total time of offloading checkpoints to CPUs and fetching them from CPUs was lower than the forward and backward pass time between two activation checkpoints in FOLD3D. Therefore, the activation checkpoint offloading costs were overlapped with computation and caused negligible impact on FOLD3D's performance.

In the evaluation, we already evaluated both typical large batch size (i.e., 2048) and small batch size (i.e., 256) as shown in Figure 12b. On the level of principle, both the activation checkpoint size and the number of FLOPs per iteration in FOLD3D are proportional to the batch size. Overall, both our and common practices [29] match the principle. In particular, the ratio between the computation time and the checkpoint offloading time remains roughly the same for all the batch sizes we evaluated. Therefore, the checkpoint offloading time remains smaller than the computation time under various batch sizes; FOLD3D's offloader causes negligible performance penalty.

Training Convergence. FOLD3D is designed to maintain the same training convergence with the baselines and to remain transparent (§3) to the training workload. Still, in Figure 14, to verify the training convergence of FOLD3D, we trained GPT-3 39B using both FOLD3D and Megatron-SP on 128 A100 GPUs. We kept the same training parameters (learning rate, batch size, and random seeds) for both systems. The results show that FOLD3D achieved the same convergence curve with Megatron, although FOLD3D achieved an obviously better loss reduction, because FOLD3D finishes each training iteration faster.

6.5 Lessons Learned

FOLD3D has two limitations. First, same as Megatron [8], [28], our system is mainly designed for large DNN models with a repeated, stacked structure. Nevertheless, compared to baselines' papers (with one or two large models evaluated), we have evaluated all the five notable and typical large models with repeated blocks. FOLD3D and all existing 3D parallel training systems (Megatron [8], [28] and DSpeed3D [19]) are currently not designed to support



(a) Reaching the same training loss 5.0, Fold3D takes 23.7% less time than Megatron (training time reduced by FOLD3D overall matched its throughput imporvement in Table 2).

(b) FOLD3D kept the same logical convergence efficiency with Megatron. The grey curve consists of the blue curve of FOLD3D and the yellow curve of Megatron.

Fig. 14: Training loss curve of GPT-3 39B using Fold3D and Megatron on 128 A100 GPUs over (a) physical time and (b) logical training steps.

DNN models with heterogeneous layers (i.e., layers which do not have the same structure or input tensor shape). For instance, all 3D parallel systems including FOLD3D are not suitable for ResNet models, because the layers in a ResNet model differ in structures and input tensor shapes. A typical ResNet model reduces the input tensor shape and increases the number of convolution filters layer by layer. FOLD3D is not suitable for DNNs with heterogeneous layers due to two reasons. The first reason, which also applies to Megatron and DSpeed3D, is that the heterogeneity easily leads to unbalanced computation across pipeline stages. When splitting models like ResNet to pipeline stages, the last stage with a linear layer will have extremely heavyweight computation compared to other stages. The unbalance makes 3D parallelism fundamentally unsuitable for DNNs with heterogeneous layers. Researchers may need to use the other two parallel dimensions or invent a new parallel dimension to replace the pipeline parallel dimension. We leave this open problem for future work. The second reason specifically for FOLD3D is that the segment slicing of FOLD3D requires the segments to be homogeneous so that a segment's computational and communicational tasks can align with those of other segments to maximize FOLD3D's effectiveness. The second limitation is that the CPU offloading mechanism in FOLD3D can consume extra CPU memory than Megatron. Fortunately, on commodity clouds, compared with GPU memory, CPU memory is cheap and extensible.

We believe FOLD3D and Megatron are complementary to each other. Megatron is optimized for training on dedicated ultra network clusters [59]. On such dedicated clusters, we envision that FOLD3D's gain over Megatron will decrease, because these clusters' Tbps network seems not to be a bottleneck. After all, FOLD3D is designed for high-performance training of large models on commodity clouds for a wide range of users, labs, and enterprises (who do not have access to these dedicated clusters). FOLD3D's much-improved throughput on commodity clouds (on many GPUs and sub-100Gbps networks) has shown its value on saving these folks' massive financial resources and natural energy. Moreover, the available network on commodity clouds is often not as large as claimed by the cloud provider. When we ran our experiments on V100-100G in a dedicated, quiet AWS cluster, by network monitoring, we found that the peak network bandwidth for each AWS tenant (us) could be only 70 to 80Gbps (see §6.3), making FOLD3D especially desirable.

7 RELATED WORK

There are tremendous systems that study the parallel techniques for DNN training, along data parallelism [19], [29], [60], [61], [62], pipeline parallelism [10], [11], [13], [20], [21], [22], [43], [63], [64], and tensor parallelism [25], [65], [66], [67], until the emergence of 3D parallel training systems [8], [9], [17]. Based on the above three foundational dimensions, there are various emerging parallelism techniques including optimizer parallelism (e.g., DeepSpeed Zero [29]), token parallelism (e.g., TeraPipe [64]), sequence parallelism [28], etc. All these various techniques are complementary to 3D parallelism with each targeting at extreme cases of large DNN training (e.g., token parallelism is for extremely long sequence training). In this paper, we focus on optimizing the 3D parallel dimensions, which serve as the foundation for today's large DNNs to scale efficiently to billions of parameters.

Systems for data parallel training. Data parallelism [19], [29], [60], [61], [62], [68] is widely adopted for distributed DNN training. Some data parallel training systems like P3 [69] and TicTac [40] adopt priority scheduling in data parallel training to overlap the data parallel communication with both forward and backward computation. Gradients of front layers are scheduled ahead of rear layers to maximize overlapping. BytePS [70] unifies all-reduce and parameter servers to utilize heterogeneous resources in a cluster. ZeRO [29] reduces the memory usage of data parallelism by sharding model parameters and gradients across GPUs. Compared with P3 and TicTac which only work for pure data parallel training, Fold3D further tackles the challenges to overlap communication with computation in 3D parallel training, which are non-trivial as we discussed in §1. Fold3D can incorporate techniques like BytePS.

Systems for pipeline parallel training. Pipeline parallelism [10], [11], [13], [20], [21], [22], [43], [63], [64], [71], [72], [73], [74] is commonly used for training large DNN models. TeraPipe [64] performs fine-grained pipeline parallelism across tokens in a single training sequence for Transformer-based models. vPipe [22] balances the memory usage and computation across pipeline stages. HetPipe [10] supports training on a set of heterogeneous GPUs with pipeline parallelism. These optimizations sorely in the pipeline parallelism dimension are orthogonal to Fold3D. When combining existing pipeline parallel systems with data parallelism, none of them can overlap data parallel communication with computation, and the data parallel communication of these systems is serialized after pipeline parallel computation.

Automatic partitioning. FlexFlow [75], Placeto [76], REGAL [77], Alpa [17] and Piper [16] automatically partition a model over multiple devices through transforming the parallelization optimization problem into a cost minimization problem. Among these works, Piper efficiently finds a near-optimal strategy for 3D parallelization combined with memory-saving techniques. However, these works focus on finding an optimal parallelization configuration, while Fold3D proposes a new 3D parallel scheduling. Note that

both FOLD3D and Megatron used the 3D configuration strategy produced by Piper (§6). We believe FOLD3D and Piper are orthogonal.

OOO [21] proposes a new training task splitting paradigm that splits the gradient computations of output and weights in the backward propagation, so that a smaller bubble size during pipelining can be achieved. However, OOO achieves this at a cost of larger memory overhead because it requires a longer duration of all layers' gradient outputs (ΣO) stashing in the GPU memory. OOO is not designed for 3D parallelism, because when OOO is combined with TP, the ΣO will be explosive as each layer's gradient output is gathered from all GPUs of the TP dimension. OOO has to keep this heavy ΣO in GPU memory until all tasks of a layer's backward pass finish. Therefore, OOO is orthogonal to all 3D parallel training systems including FOLD3D.

There are also various pioneer works that target at new training paradigms based on Transformer-like models, introducing sparsely activated DNN training. Pathways [78] is a recent Multi Program Multiple Data (MPMD) training framework (proposed by Google) that runs multiple training tasks/programs (i.e., each task/program is a single SGD procedure; tasks may share parameters with each other) to fully exploit a cluster's heterogeneous GPU resources. Still, Pathways is complementary to Single Program Multiple Data systems including Megatron, DeepSpeed, and FOLD3D, because within each training task (program), the 3D parallelism technique is still essential for scaling to a large number of GPUs/TPUs with heterogeneous inter-links between devices.

Besides, Mixture-of-Expert [34], [67] extends Transformer models with many sparsely activated experts. Many emerging training systems such as FasterMoE [35] and Tutel [79] to accelerate MoE workloads. These systems are orthogonal to FOLD3D. We believe our AIAO scheduling has potentials to benefit MoE models, as the MoE training paradigms also requires the DP, PP, and TP parallel dimensions [17]. Certainly, new challenges will be encountered as MoE models introduced many asymmetric, sparsely activated computational tasks to the traditional DNN training, bring complexities to find the optimal 3D parallel scheduling. We leave this in future work.

8 CONCLUSION

We present the FOLD3D system, which maximally overlaps communication and computation tasks in 3D parallel training of large DNN models on commodity clouds. By folding a model into segments, FOLD3D conducts AIAO to achieve the an all-parallel scheduling between communication and computation tasks. FOLD3D can benefit most people who demand training and fine-tuning large DNN models.

ACKNOWLEDGMENTS

We thank all reviewers for their valuable comments. The work is supported in part by the Huawei Flagship Research Grant in 2021, the HKU-SCF FinTech Academy R&D Funding Scheme in 2021 and 2022, HK RIF (R7030-22), HK ITF (GHP/169/20SZ), and the Pujiang Lab (Heming Cui is a courtesy researcher in this lab).

REFERENCES

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[2] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlche Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf

[3] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, R. Wang, J. Gao, M. Zhou, and H.-W. Hon, "Unified language model pre-training for natural language understanding and generation," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. deBuc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/c20bb2d9a50d5ac1f713f8b34d9aac5a-Paper.pdf

[4] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 8342–8360. [Online]. Available: https://aclanthology.org/2020.acl-main.740

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=YicbFdNTTy

[6] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," *arXiv preprint arXiv:2103.14030*, 2021.

[7] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in vision: A survey," *arXiv preprint arXiv:2101.01169*, 2021.

[8] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient 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*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476209

[9] "microsoft/deepspeed," https://github.com/microsoft/DeepSpeed.

[10] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[11] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "Pipemare: Asynchronous pipeline parallel dnn training," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[12] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized stochastic gradient descent." in *NIPS*, vol. 4, no. 1. Citeseer, 2010, p. 4.

[13] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: a pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.

[14] "Nvlink," https://www.nvidia.com/en-us/data-center/nvlink/.

[15] "Infiniband and remote dma (rdma) interfaces," https://www.kernel.org/doc/html/v5.11/driver-api/infiniband.html.

[16] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for dnn parallelization," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[17] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, J. E. Gonzalez *et al.*, "Alpa: Automating inter-and intra-operator parallelism for distributed deep learning," *arXiv preprint arXiv:2201.12023*, 2022.

[18] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[19] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.

[20] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.

[21] H. Oh, J. Lee, H. Kim, and J. Seo, "Out-of-order backprop: an effective scheduling technique for deep learning," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 435–452.

[22] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[23] M. Zhu, Y. Zhuo, C. Wang, W. Chen, and Y. Xie, "Performance evaluation and optimization of hbm-enabled gpu for data-intensive applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 5, pp. 831–840, 2018.

[24] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, pp. 103–112, 2019.

[25] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *CoRR*, vol. abs/1909.08053, 2019. [Online]. Available: http://arxiv.org/abs/1909.08053

[26] "Nvidia/megatron-lm," https://github.com/NVIDIA/Megatron-LM/tree/main/megatron.

[27] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-Offload: Democratizing Billion-Scale model training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 551–564. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/ren-jie

[28] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *arXiv preprint arXiv:2205.05198*, 2022.

[29] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[32] Z. Zhang, X. Han, H. Zhou, P. Ke, Y. Gu, D. Ye, Y. Qin, Y. Su, H. Ji, J. Guan *et al.*, "Cpm: A large-scale generative chinese pre-trained language model," *AI Open*, vol. 2, pp. 93–99, 2021.

[33] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, pp. 1–67, 2020.

[34] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," 2021.

[35] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, and J. Tang, "Fastmoe: A fast mixture-of-expert training system," *arXiv preprint arXiv:2103.13262*, 2021.

[36] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[37] V. Gabeur, C. Sun, K. Alahari, and C. Schmid, "Multi-modal transformer for video retrieval," in *European Conference on Computer Vision*. Springer, 2020, pp. 214–229.

[38] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *Big Learning NIPS Workshop*, vol. 6, 2013, p. 2.

[39] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[40] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," *SysML 2019*, 2019.

[41] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476205

[42] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[43] S. Zhao, F. Li, X. Chen, X. Guan, J. Jiang, D. Huang, Y. Qing, S. Wang, P. Wang, G. Zhang *et al.*, "Vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[44] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[45] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "Hetpipe: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 307–321.

[46] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[47] W. A. Gardner, "Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique," *Signal processing*, vol. 6, no. 2, pp. 113–133, 1984.

[48] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, "A theoretical framework for back-propagation," in *Proceedings of the 1988 connectionist models summer school*, vol. 1, 1988, pp. 21–28.

[49] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[50] Nvidia, "Nccl hangs during ncclsend and ncclrecv." [Online]. Available: https://github.com/NVIDIA/nccl/issues/584

[51] "jcpeterson/openwebtext," https://github.com/jcpeterson/openwebtext.

[52] "huggingface/wikipedia," https://huggingface.co/datasets/wikipedia.

[53] "Wudaocorpora 2.0," https://resource.wudaoai.cn/home.

[54] "allenai/c4," https://huggingface.co/datasets/allenai/c4.

[55] S. Rashidi, M. Denton, S. Sridharan, S. Srinivasan, A. Suresh, J. Nie, and T. Krishna, "Enabling compute-communication overlap in distributed deep learning training platforms," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 540–553.

[56] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[57] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team, "An empirical model of large-batch training," *arXiv preprint arXiv:1812.06162*, 2018.

[58] C. Li, M. Zhang, and Y. He, "Curriculum learning: A regularization method for efficient and stable billion-scale gpt model pretraining," *arXiv preprint arXiv:2108.06084*, 2021.

[59] "Nvidia selene: Leadership-class supercomputing infrastructure," https://www.nvidia.com/en-us/on-demand/session/supercomputing2020-sc2019/.

[60] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[61] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[62] Y. Xu, H. Lee, D. Chen, H. Choi, B. Hechtman, and S. Wang, "Automatic cross-replica sharding of weight update in data-parallel training," *arXiv preprint arXiv:2004.13336*, 2020.

[63] L. Guan, W. Yin, D. Li, and X. Lu, "Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training," *arXiv preprint arXiv:1911.04610*, 2019.

[64] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6543–6552.

[65] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni *et al.*, "Gspmd: general and scalable parallelization for ml computation graphs," *arXiv preprint arXiv:2105.04663*, 2021.

[66] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," *Advances in neural information processing systems*, vol. 31, 2018.

[67] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[68] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf *et al.*, "Mlperf training benchmark," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 336–349, 2020.

[69] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 132–145, 2019.

[70] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.

[71] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[72] W. Zeng, X. Ren, T. Su, H. Wang, Y. Liao, Z. Wang, X. Jiang, Z. Yang, K. Wang, X. Zhang *et al.*, "Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation," *arXiv preprint arXiv:2104.12369*, 2021.

[73] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "Pipetransformer: Automated elastic pipelining for distributed training of transformers," *arXiv preprint arXiv:2102.03161*, 2021.

[74] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino, "Efficient algorithms for device placement of dnn graph operators," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 451–15 463, 2020.

[75] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *SysML 2019*, 2019.

[76] S. Bojja Venkatakrishnan, S. Gupta, H. Mao, M. Alizadeh *et al.*, "Learning generalizable device placement algorithms for distributed machine learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[77] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals, "Reinforced genetic algorithm learning for optimizing computation graphs," *arXiv preprint arXiv:1905.02494*, 2019.

[78] P. Barham, A. Chowdhery, J. Dean, S. Ghemawat, S. Hand, D. Hurt, M. Isard, H. Lim, R. Pang, S. Roy *et al.*, "Pathways: Asynchronous distributed dataflow for ml," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 430–449, 2022.

[79] C. Hwang, W. Cui, Y. Xiong, Z. Yang, Z. Liu, H. Hu, Z. Wang, R. Salas, J. Jose, P. Ram *et al.*, "Tutel: Adaptive mixture-of-experts at scale," *arXiv preprint arXiv:2206.03382*, 2022.

Fanxin Li received the BE degree from Xi'an Jiaotong University in 2019. He is currently working toward the PhD degree at The University of Hong Kong. His research interests include distributed machine learning and cloud computing.

Shixiong Zhao received his Bachelor degree in HKU and his master degree in HKUST. He is currently a PhD student in Computer Science of HKU. He is under the supervision of Prof. Heming Cui. His research interests include distributed systems for high performance computing, distributed systems and system security. He is a student member of IEEE.

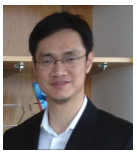
Yuhao Qing received his Bachelor degree in City University of Hong Kong. He is currently a PhD student in Computer Science of HKU, under the supervision of Prof. Heming Cui. His research interests includes machine learning systems and cloud computing.

Xusheng Chen received his Bachelor degree in HKU. He is currently a Ph.D. student in Computer Science of HKU. He is under the supervision of Prof. Heming Cui. His research interests include distributed consensus protocols, distributed systems and system security.

Xiuxian Guan received his bachelor's degree in the University of Science and Technology of China. He is currently a PhD student in the Department of Computer Science in HKU, co-supervised by Prof. Heming Cui from HKU and Prof. Rui Wang from SusTech. His research interest includes distributed systems, wireless networks, machine learning and more.

Sen Wang received the B.S. degree from the University of Science and Technology of China (USTC), Hefei, China, in 2005, the M.S. degree from the Chinese Academy of Sciences (CAS), Beijing, China, in 2008, and the Ph.D. degree from Tsinghua University, Beijing, China, in 2014, all in computer science. From 2014 to 2019, he was a lecturer and then an associate professor at Chongqing University, Chongqing, China. Currently, he is a senior researcher at Huawei, Hongkong. His research interests include information-centric networking, Federated Learning and AI for System.

Gong Zhang is a chief architect researcher scientist, director of the Huawei Future Network Theory Lab. His major research directions are network architecture and large-scale distributed systems. He has abundant experience on system architect in networks, distributed system and communication system for more than 20 years. He has more than 90 global patents.



Heming Cui is an Associate Professor in Computer Science of HKU. His research interests include operating systems, programming languages, distributed systems, and cloud computing, with a particular focus on building software infrastructures and tools to improve reliability and security of real-world software. He is a member of IEEE.