Efficient Pipeline Planning for Expedited Distributed DNN Training

Ziyue Luo*, Xiaodong Yi*, Guoping Long†, Shiqing Fan†, Chuan Wu*, Jun Yang†, Wei Lin†
*Department of Computer Science, The University of Hong Kong, Email: {zyluo, xdyi, cwu}@cs.hku.hk
†Alibaba, Email: longguoping@gmail.com, {shiqing.fsq, muzhuo.yj, weilin.lw}@alibaba-inc.com

***Abstract*—To train modern large DNN models, pipeline parallelism has recently emerged, which distributes the model across GPUs and enables different devices to process different microbatches in pipeline. Earlier pipeline designs allow multiple versions of model parameters to co-exist (similar to asynchronous training), and cannot ensure the same model convergence and accuracy performance as without pipelining. Synchronous pipelining has recently been proposed which ensures model performance by enforcing a synchronization barrier between training iterations. Nonetheless, the synchronization barrier requires waiting for gradient aggregation from all microbatches and thus delays the training progress. Optimized pipeline planning is needed to minimize such wait and hence the training time, which has not been well studied in the literature. This paper designs efficient, near-optimal algorithms for expediting synchronous pipeline-parallel training of modern large DNNs over arbitrary inter-GPU connectivity. Our algorithm framework comprises two components: a pipeline partition and device mapping algorithm, and a pipeline scheduler that decides processing order of microbatches over the partitions, which together minimize the per-iteration training time. We conduct thorough theoretical analysis, extensive testbed experiments and trace-driven simulation, and demonstrate our scheme can accelerate training up to 157% compared with state-of-the-art designs.**

I. Introduction

Deep learning has advanced various applications in a wide range of domains [1][2][3]. Deep Neural Networks (DNNs) have significantly grown in size in recent years, in pursuit of better model accuracy. Training of large models over large datasets has promoted the rapid development of distributed DNN training frameworks (e.g., TensorFlow [4], PyTorch [5]).

A number of parallel-training paradigms have been adopted in practice. Data parallelism [6] partitions the training dataset among workers. Each worker holds a copy of the DNN, computes parameter updates using the local dataset, and synchronizes parameter updates with others periodically. AllReduce operation [7] is a common approach for parameter synchronization among workers. To handle large models which cannot be fit entirely into a single device's memory, model parallelism [8] partitions a DNN model and places model partitions on different devices. In each training iteration, a mini-batch is processed by model partitions on the devices one after another, through forward propagation followed by backward propagation. Such vanilla model parallelism suffers from low device utilization, as only one device is active at each time when a mini-batch is trained across the devices hosting different model partitions. Pipeline parallelism [9] has been proposed to maximize device utilization during training. Similar to model parallelism, it partitions a DNN into stages and places stages over multiple devices; it further partitions each mini-batch of training data into equal-sized microbatches, and allows different devices to process different microbatches at the same time (*i.e.*, microbatch pipelining).

Most works on pipeline parallelism [10][11][12][13] adopt asynchronous pipelining, by injecting microbatches into the training pipeline one by one and updating model parameters with gradients computed with a microbatch, whenever its backward propagation is done. Asynchronous pipeline parallelism maximizes GPU utilization by fully saturating the pipeline. However, the processing of different microbatches overlaps, each updating the model using gradients computed based on outdated parameters that are learned on different earlier microbatches, which may inevitably slow down or even prevent training convergence, and render a model whose accuracy differs from that trained without pipelining [14].

To ensure model convergence and accuracy, synchronous pipeline parallelism has been advocated by a few recent studies [15][16]. It enforces a synchronization barrier between training iterations, to aggregate gradients computed with all microbatches before applying them for model update. Such a synchronization barrier flushes the pipeline and introduces waiting time (for training completion of all microbatches) into each training iteration, leading to lower device utilization as compared to asynchronous pipeline training. Optimal planning of synchronous pipeline training is needed to improve device utilization and minimize per-iteration training time, to achieve similar training time as asynchronous pipelining while providing convergence and accuracy guarantees. Pipeline planning includes DNN model partition, replication and device placement, as well as scheduling the order of microbatch processing across the devices within each training iteration. Non-trivial challenges exist, as follows:

First, in a typical DNN model, layers are not uniform in terms of computation time, parameter size and activation size. Optimal model partition over devices is hence challenging.

Second, previous pipeline designs have been restricted to homogeneous inter-GPU connectivity (or homogeneous in each level of a hierarchical topology) [15][10]. GPU interconnectivity is often more complicated in a practical machine learning (ML) cluster, including PCI-e or NVLink within

This work was supported in part by Alibaba Group through Alibaba Innovative Research (AIR) Program, and grants from Hong Kong RGC under the contracts HKU 17204619, 17208920 and 17207621.

a physical server [17], RDMA or TCP network between servers [18]. We will show that heterogeneous GPU interconnectivity leads to an exponential number of solutions for DNN model partition and device mapping (Sec. IV-B), adding to the difficulty of finding efficient, near-optimal solutions.

Third, deciding the execution order of all microbatches over all devices, respecting inter-stage dependencies and minimizing per-iteration training time, falls in the category of job shop problems. The job shop problem is NP-hard [19] even with only two machines (aka GPUs in our case).

Tackling the challenges, we design near-optimal algorithms that efficiently partition a given DNN model, replicate and distribute the partitions over available GPUs with arbitrary inter-GPU connectivity, and schedule microbatch processing over the stages to minimize per-iteration training time. Our main techniques and contributions are summarized as follows:

▷ Assuming model partition and device mapping are given, we design an efficient list ordering method to decide the processing order of microbatches on different GPUs, and then a scheduling algorithm that minimizes idle time of devices based on the order. With thorough theoretical analysis, we identify an upper bound of per-iteration training time, decided by two key factors: the number of stages that the DNN is partitioned into, and the maximum time to process a microbatch on a single stage or an inter-stage communication channel.

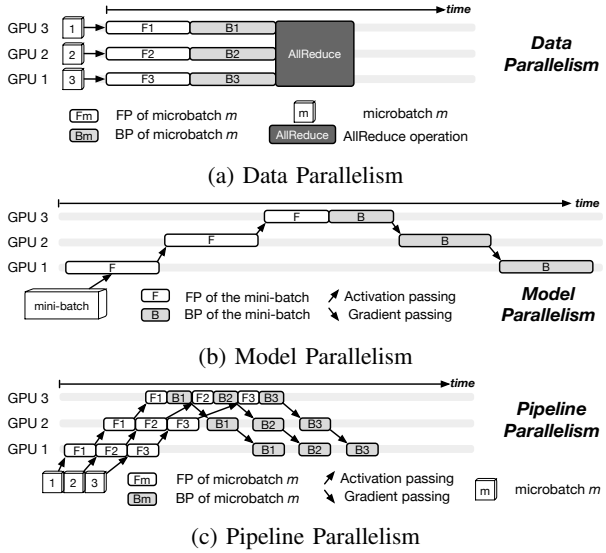
▷ We are hence inspired to design a pipeline partition and device mapping algorithm to minimize the maximum per-stage/channel execution time, given the number of stages to partition the model into. A recursive min-cut method is designed to identify a device order that maximizes inter-GPU bandwidth utilization. Based on the device order, we use dynamic programming to derive the optimal partition, replication and mapping solution.

▷ Our complete synchronous pipeline planning algorithm, *SPP*, iteratively invokes the pipeline partition/mapping algorithm and the execution order scheduler to identify the best number of partitioned stages and the set of near-optimal pipeline execution strategies accordingly. We rigorously analyze *SPP* and prove a worst-case performance guarantee.

▷ We conduct extensive testbed experiments and trace-driven simulation, carefully comparing *SPP* with state-of-the-art pipeline-training paradigms, including GPipe [15], PipeDream [10] and HetPipe [12]. Experimental results show that *SPP* can accelerate training up to 147% compared to GPipe, 157% to PipeDream and 80% to HetPipe in terms of per-iteration training time, and achieves the target accuracy in the most expedited manner as compared to baselines. We observe that *SPP* can strike a balance between the number of stages and the maximum per-stage execution/communication time in DNN partition, and maximally overlap communication and computation with its pipeline execution scheduling.

II. Background and Related Work

DNN Training. A DNN model comprised of multiple layers is usually trained over a large dataset iteratively to minimize a loss function [20]. The dataset is typically divided into



(a) Data Parallelism

(b) Model Parallelism

(c) Pipeline Parallelism

Fig. 1: Data parallelism vs. model parallelism vs. pipeline parallelism: 1 mini-batch divided into 3 microbatches.

equal-sized mini-batches. In each training iteration, one mini-batch is processed to update the DNN model as follows: (1) *forward propagation (FP)*: the mini-batch is computed by each layer of the DNN sequentially to derive a loss; 2) *backward propagation (BP)*: gradients of model parameters are computed based on the loss from the last layer to the first layer; 3) a *gradient update* operation applies computed gradients to parameters in each layer with an optimization algorithm, *e.g.*, stochastic gradient descent (SGD) or adaptive moment estimation (Adam) [20].

DNN Model Partition and Device Mapping. A number of studies have focused on partition and device mapping strategies for large DNN models through learning-based methods [21][22][23], which require large computing resources and long training time to derive a satisfying policy for one training job. A few efforts [24][25] exploit efficient heuristics for DNN model partition and device mapping at the operation level, requiring detailed cost modeling of the DNN model and accurate profiling of operation execution time. Our work focuses on layer-level DNN model partitioning and mapping, and derives a polynomial-time pipeline planning strategy.

Data Parallelism (DP) and Model Parallelism (MP) are commonly adopted to parallelize training across multiple devices. As shown in Fig. 1(a), with DP, three mini-batches are each trained on one GPU with a complete copy of model parameters; an AllReduce operation synchronizes computed gradients after training of all mini-batches. With MP (Fig. 1(b)), in each training iteration, a mini-batch is fed into the device hosting the first stage(s) of the DNN model for FP, and the computed activations are passed to later stages on other devices for FP; during BP, gradients are computed and passed from one device to another following reverse sequence of the stage(s). In this way, only one device is active at each time, where FP or BP of the mini-batch is being carried out, while other devices are idle, leading to low device utilization.

Pipeline Parallelism Based on model parallelism, *pipeline parallelism* further divides a mini-batch into equal-sized microbatches (Fig. 1(c)). The microbatches are consecutively fed into the device hosting the first stage(s) whenever the forward computation of the previous microbatch is done on this device, rendering a training pipeline. Consequently, it enables multiple devices to process different microbatches simultaneously.

Asynchronous Pipeline Training. PipeDream [10] partitions a DNN model over multiple servers, allowing stage replication among the servers, and further divides a stage over GPUs within each server, aiming at minimizing the maximum time to process a single stage. Its server configuration and inter-server connectivity are both homogeneous. Stage execution is scheduled to ensure that every FP stage is immediately followed by a BP stage. Geng *et al.* [11] study pipeline parallelism over heterogeneous GPUs, and propose a dynamic tuning algorithm to identify straggler devices and redistribute the DNN model for better load balance. In HetPipe [12], each node (comprised of homogeneous GPUs) trains the DNN model in a pipelined manner similar to PipeDream without stage replication; DP is used for training and parameter synchronization among nodes. With asynchronous pipelining, microbatches are trained on outdated versions of model parameters to compute gradients, leading to slow model convergence and lower accuracy of the obtained model as compared to synchronous training [14]. Several studies have investigated mitigating the accuracy loss via weight prediction [26] or randomized smoothing [27], under restricted assumptions of training loss functions.

Synchronous Pipeline Training. GPipe [15] is a synchronous pipeline training framework, including (1) a partition strategy that ensures approximately the same number of DNN layers on each GPU, and (2) a schedule to execute all FP before starting any BP. It does not allow stage replication and provides no device mapping strategies. We design efficient algorithms to deal with all aspects of synchronous pipeline planning.

III. System Model

A. DNN Model and Device Connectivity

We consider a DNN model, $\mathcal{D}$, consisting of L layers, *e.g.*, attention layers, convolutional and fully-connected layers. In each training iteration, a mini-batch is divided into M equal-sized microbatches of size Z each. Every microbatch is trained through FP through all L layers, followed by BP over the L layers in the reverse order. We divide $\mathcal{D}$ into multiple *stages*, place the stages on different GPUs, and allow different GPUs to process different microbatches simultaneously in the pipelined manner. Following the end of BP of all microbatches, a gradient aggregation operation aggregates gradients computed from all microbatches and applies them to update the model parameters. As time needed for gradient aggregation and apply is much shorter than FP/BP time , we ignore it in our pipeline parallelism design.

V homogeneous GPUs on multiple physical servers are available for training this DNN model.[1] We consider a variety of GPU inter-connectivity, including PCIe or NVLink (providing direct GPU-GPU communication channel) between GPUs in the same physical server (*e.g.*, in NVIDIA DGX-1 [17]), TCP or RDMA connections across GPUs in different servers [18], with various bandwidth levels. Graph $G(\mathcal{V}, \mathcal{E})$ represents the multi-GPU system for training $\mathcal{D}$, where $\mathcal{V}$ includes the V GPUs and $\mathcal{E}$ contains all inter-connections between the GPUs. Each edge (v, v') in $\mathcal{E}$ is associated with a weight, $b_{v,v'}$, representing the available bandwidth between GPU v and GPU v'. Let b_{min} and b_{max} be the minimum and maximum bandwidth among all edges in $\mathcal{E}$, respectively.

The forward (backward) computation time of a microbatch over layer l of DNN $\mathcal{D}$ on a given GPU is p_l^f (p_l^b). Let α_l be the size of parameters/gradients of layer l, which can be profiled through one trial run of the whole model using several training iterations. $d_{l,l+1}^f$ denotes the size of activations passed from layer l to layer $l+1$ during FP, and $d_{l+1,l}^b$ is the size of gradients transferred from layer $l+1$ to layer l during BP.

B. Synchronous Pipeline with Stage Replication

We target synchronous pipeline design that minimizes per-iteration training time, including two subproblems:

1) **Pipeline Partition, Replication and Device Mapping:** We decide the stage set $\mathcal{S} = \{s_1, s_2, \ldots, s_{|\mathcal{S}|}\}$ with $|\mathcal{S}| \leq V$ to partition model $\mathcal{D}$ into, and a device mapping function $\mathcal{F}$: $\mathcal{S} \rightarrow \mathbb{P}(\mathcal{V})$, where $\mathbb{P}(\mathcal{V})$ includes all subsets of device set $\mathcal{V}$.

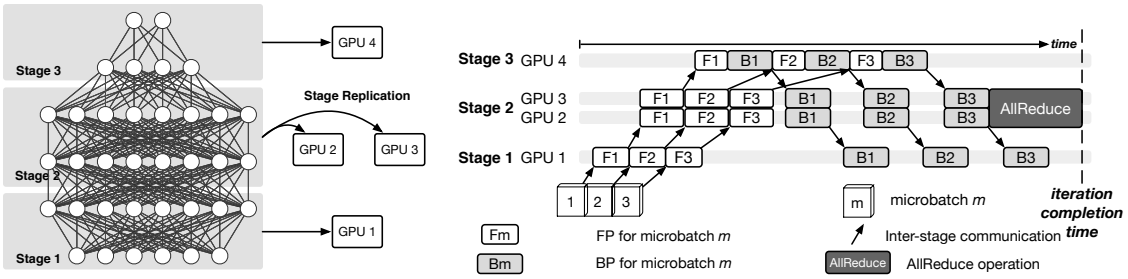
We consider classical *interval partition*, that each stage consists of a number of consecutive layers: for stage $s \in \mathcal{S}$, if layer l_{start} and layer l_{end} belong to s, then all layers l, with $l_{start} \leq l \leq l_{end}$, belong to s. Without loss of generality, we assume a sequential dependency through $s_1, s_2, \ldots, s_{|\mathcal{S}|}$, *i.e.*, the last layer l_{end}^n in stage s_n is the predecessor of the first layer l_{start}^{n+1} in stage s_{n+1} in the DNN model.

$\mathcal{F}$ maps each stage $s \in \mathcal{S}$ to one or multiple GPUs, ensuring that each GPU hosts exactly one stage or one replica of a stage. In our design, we allow a stage to be replicated and executed over multiple GPUs in a data-parallel fashion. Suppose stage s is replicated over k GPUs $\{v_1, v_2, \ldots, v_k\}$. Processing of a microbatch by stage s is distributed over the k GPUs (by evenly dividing input data among these GPUs), and we assume that the forward (backward) computation time of each layer l in stage s on each replica device is now $\frac{p_l^f}{k}$ ($\frac{p_l^b}{k}$).[2] Fig. 2(a) gives an example, where a 6-layer DNN model is trained using 4 GPUs. The model is partitioned into three stages with stage 2 replicated over GPU 2 and GPU 3.

Such stage replication may improve GPU utilization and further balance stage processing time, together with stage partition. In Fig. 2, supposing the size of layers in stage 2 is much larger than the other stages, replicating stage 2 on two GPUs allows forward/backward computation time of the stage to be similar to others, as shown in Fig. 2(b).

[1] Training a DNN using GPUs of the same type is the norm in today's production systems, based on our discussions with leading AI cloud operators.

[2] We note that non-linear change of training time may happen when a layer is replicated, i.e., each layer replica's execution time is not exactly $\frac{1}{k}$ of the layer's processing time without input data partition. Our algorithm can be readily extended to the non-linear case by modeling the computation time of each layer given different input data sizes.



(a) Stage partition, replication and device mapping

(b) Execution schedule of 3 microbatches

Fig. 2: A pipeline parallelism design example

TABLE I: NOTATION

$\mathcal{D}$	the DNN model
L	# of layers
M	# of microbatches in one iteration
V	# of GPUs
$\mathcal{G}(\mathcal{V},\mathcal{E})$	GPU inter-connection graph ($\mathcal{V}$: GPUs; $\mathcal{E}$: inter-GPU connections)
$b_{v,v'}$	bandwidth between GPU v and GPU v'
$b_{min}(b_{max})$	minimum (maximum) bandwidth in $\mathcal{E}$
$p_l^f\,(p_l^b)$	FB (BP) computation time of layer l per microbatch
α_l	size of parameters (gradients) of layer l
$d_{l,l+1}^f\ (d_{l+1,l}^b)$	size of activations (gradients) from layer l to $l+1$ ($l+1$ to l) during FP (BP)
$\mathcal{S}$	set of all stages that $\mathcal{D}$ is partitioned into
$\mathcal{S}_{repl}$	set of replicated stages
$\mathcal{F}:\mathcal{S}\rightarrow\mathbb{P}(\mathcal{V})$	device mapping function from stages to sets of GPUs
$c_{s_n,s_{n+1}}^f$ (c_{s_{n+1},s_n}^b)	communication time between stages s_n and s_{n+1} during FP (BP)
$e_{m,s_n}^f(e_{m,s_n}^b)$	start time of stage s_n's processing of microbatch m during FP (BP)
A_s	time taken by AllReduce operation of stage s
e_s^A	start time of AllReduce operation of stage s

After completion of backward computation of microbatches on all k replicas of stage s, a ring AllReduce operation [7] synchronizes gradients of stage s across the k GPUs. Especially, the k GPUs are organized into a logical ring topology, and each GPU exchanges gradients/parameters with its neighbors in the ring through inter-GPU connections. The size of communication data (gradients and parameters) involved in the AllReduce operation is $\frac{2(k-1)}{k}\sum_{l\in s}\alpha_l$ per GPU [28]. The time taken by the AllReduce operation, denoted by A_s, is further decided by the minimum connection bandwidth among the k GPUs:

$$A_s = \frac{2(k-1)\sum\limits_{l\in s}\alpha_l}{k\min\limits_{v,v'\in\{v_1,v_2,\ldots,v_k\}} b_{v,v'}} \tag{1}$$

2) **Execution Scheduling:** We also decide the execution order of processing each microbatch on each stage, as well as running the AllReduce operations for replicated stages. Let e_{m,s_n}^f (e_{m,s_n}^b) be the start time of forward (backward) computation of microbatch m on stage s_n.

Execution schedule should respect forward-backward dependency and stage dependency. Each GPU can only process one microbatch at a time. Let $c_{s_n,s_{n+1}}^f$ and c_{s_{n+1},s_n}^b represent the inter-stage communication time between stage s_n and stage s_{n+1} during FP and BP, respectively. We ignore the time for data passing between layers residing in the same GPU. We formulate the dependencies as follows.

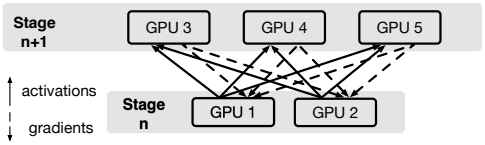


Fig. 3: Inter-stage communication: an example

- (*Forward-backward dependency*):

$$e_{m,s_{|\mathcal{S}|}}^f + \frac{\sum\limits_{l\in s_{|\mathcal{S}|}} p_l^f}{|\mathcal{F}(s_{|\mathcal{S}|})|} \le e_{m,s_{|\mathcal{S}|}}^b, \forall m\in\{1,\ldots,M\}$$

- (*Stage dependency*):

$$e_{m,s_n}^f + \frac{\sum\limits_{l\in s_n} p_l^f}{|\mathcal{F}(s_n)|} + c_{s_n,s_{n+1}}^f \le e_{m,s_{n+1}}^f,$$
$$\forall m\in\{1,\ldots,M\}, n\in\{1,\ldots,|\mathcal{S}|-1\}$$
$$e_{m,s_n}^b + \frac{\sum\limits_{l\in s_n} p_l^b}{|\mathcal{F}(s_n)|} + c_{s_n,s_{n-1}}^b \le e_{m,s_{n-1}}^b,$$
$$\forall m\in\{1,\ldots,M\}, n\in\{2,\ldots,|\mathcal{S}|\}$$
$$e_{1,s_1}^f = 0$$

To compute inter-stage communication time, when s_n and/or s_{n+1} are replicated over multiple GPUs, we evenly distribute the data being transmitted across inter-stage links. For example in Fig. 3, s_n is replicated onto 2 GPUs and s_{n+1} onto 3 GPUs. During FP, 1/3 of the activations derived by each of the two GPUs hosting stage n is sent to each of the three GPUs hosting stage $n+1$. During backward propagation, each GPU running s_{n+1} splits its gradients (computed with a microbatch) into two sets of gradients corresponding to two smaller batches, and sends the two sets to the two GPU of s_n, respectively; each replica of s_n sums up received gradients from replicas of s_{n+1}. The inter-stage communication time is decided by the minimum link bandwidth between GPUs in $\mathcal{F}(s_n)$ and in $\mathcal{F}(s_{n+1})$: (note $d_{l_{end}^n,l_{start}^{n+1}}^f$ and $d_{l_{start}^{n+1},l_{end}^n}^b$ are data size produced by an entire microbatch, *i.e.*, sum of data produced by all replicas of a stage)

$$c_{s_n,s_{n+1}}^f = \frac{d_{l_{end}^n,l_{start}^{n+1}}^f}{|\mathcal{F}(s_n)||\mathcal{F}(s_{n+1})|\min\limits_{v\in\mathcal{F}(s_n),v'\in\mathcal{F}(s_{n+1})} b_{v,v'}}$$

$$c^b_{s_{n+1},s_n} = \frac{d^b_{l^{n+1}_{start},l^n_{end}}}{|\mathcal{F}(s_n)||\mathcal{F}(s_{n+1})| \min\limits_{v\in\mathcal{F}(s_n),v'\in\mathcal{F}(s_{n+1})} b_{v',v}}$$

Let $\mathcal{S}_{repl}$ be the set of replicated stages, and e^A_s denote the start time of AllReduce operation of stage $s \in \mathcal{S}_{repl}$. We further have:

- (*AllReduce operation dependency*):

$$e^b_{m,s} + \frac{\sum\limits_{l\in s} p^b_l}{|\mathcal{F}(s)|} \leq e^A_s, \forall m \in \{1,\ldots,M\}, s \in \mathcal{S}_{repl}$$

We aim at minimizing the makespan of training all M microbatches, *i.e.*, the per-iteration training time of the DNN:

$$\text{minimize } \max\{\max_{m\in\{1,2,\ldots,M\}}(e^b_{m,s_1} + \frac{\sum\limits_{l\in s_1} p^b_l}{|\mathcal{F}(s_1)|}), \max_{s\in\mathcal{S}_{repl}}(e^A_s + A_s)\} \tag{2}$$

An AllReduce operation and inter-stage communication do not share the same inter-GPU connections: the former uses links between GPUs hosting replicas of the same stage, while the latter is between GPUs hosting different stages (also recall that one GPU can only host (a replica of) one stage). The AllReduce operation of a replicated stage s_n can take place at the same time as inter-stage communication and backward computation of stages $s_{n-1}, s_{n-2}, \ldots$, as well as AllReduce operations of other replicated stages. Therefore, the completion time of a training iteration in (2) is decided by the latest among backward computation completion time of all microbatches over stage 1 and end time of AllReduce operations of all replicated stages. Note that our schedule may not ensure that microbatches are processed in the same sequence at each stage; instead, we enforce a synchronization barrier in each training iteration, as represented by the inside $\max$ over all microbatches in (2).

An example execution schedule is given in Fig. 2(b). As stage 2 is replicated over two GPUs, an AllReduce operation happens when backward computation of all three microbatches over both GPUs has been done, ensuring the model parameters on GPU 2 and GPU 3 are updated the same.

IV. Pipeline Planning Algorithms

We now design algorithms for efficient synchronous pipeline training. We start with execution scheduler design, assuming model partition and device mapping are given; then we devise the partition and device mapping algorithm that minimizes per-iteration training time together with the execution scheduler.

A. Execution Scheduler

Given model partitions $\mathcal{S}$ and device mapping $\mathcal{F}$, our scheduling problem, as presented in Sec. III-B2, is a special case of the NP-hard job shop problem [19]: microbatches correspond to jobs of the same type and stages correspond to machines in the job shop problem, and the objective is to minimize the total time of executing all jobs. We design an efficient *pipeline execution* (PE) scheduling algorithm to achieve a proven performance bound.

The PE algorithm contains two modules: 1) an ordering method to decide execution order of microbatches over stages on different GPUs, and 2) an algorithm that schedules pipeline execution based on the computed order.

1) Execution ordering: We define a *computation block* as the forward or backward computation of a stage. As backward computation of the last stage $s_{|\mathcal{S}|}$ follows immediately forward computation of $s_{|\mathcal{S}|}$, we merge stage $s_{|\mathcal{S}|}$'s forward and backward computation blocks into a single computation block. We define the inter-stage communication from s_n to s_{n+1} or from s_{n+1} to s_n to be a *communication block*, including all communication over this *communication channel*, *i.e.*, the set of connections from GPU(s) hosting the former stage to GPU(s) hosting the latter stage. The end-to-end training of every microbatch in a training iteration involves $2|\mathcal{S}|-1$ computation blocks, $2|\mathcal{S}|-2$ communication blocks and $|\mathcal{S}_{repl}|$ AllReduce operations for replicated stages. Let $\mathcal{J} = \{1,2,\ldots,4|\mathcal{S}|-3\}$ be the ordered list of all computation and communication blocks, with blocks ordered according to their execution dependencies.

An execution order queue, U_s, is maintained for each stage $s \in \mathcal{S}$, containing (`microbatch index`, `block number`) pairs indicating the order of processing microbatches by forward or backward computation blocks of stage s.

For each block $j \in \mathcal{J}$, we maintain an *available microbatch* queue Q_j, containing microbatches which have been processed by block $j-1$ but not by j. Initially Q_1 includes all microbatches in order of their indices, and $Q_j = \emptyset, \forall j \in \mathcal{J}/\{1\}$.

We order microbatch processing over the blocks as follows. According to the order of blocks in $\mathcal{J}$, we pop out one microbatch m at the head of a non-empty queue Q_j, and push it to the end of queue Q_{j+1} of the next block (if j is not the last block in $\mathcal{J}$); if block j is a computation block of stage s, we add (m,j) to execution order queue U_s. Going through the block list, we identify at most one microbatch to be processed by each block, corresponding to microbatches that can be processed about simultaneously. We loop through the block list repeatedly until all available microbatch queues are empty ($Q_j = \emptyset, \forall j \in \mathcal{J}$), *i.e.*, end-to-end training of all microbatches is ordered.

2) Scheduling: We next exploit the execution order queues, U_s's, and schedule a microbatch's processing on a block as soon as it is ready. We start by popping the first (microbatch index, block number) out of queue U_{s_1} of the first stage s_1, and process the corresponding microbatch on the respective block. Once a computation block is executed, the successor communication block is immediately run (upon the communication channel becoming idle). Upon processing completion of a scheduled computation block of stage s or a communication block which transmits data to stage s, we examine queue U_s: if the first (microbatch index, block number) in U_s is ready to be executed (*i.e.*, the microbatch has been processed by the precedent block), we pop it out and run it. This procedure terminates when $U_s = \emptyset, \forall s \in \mathcal{S}$, *i.e.*, all microbatches have been processed by all computation and communication blocks.

For each replicated stage $s \in \mathcal{S}_{repl}$, when all microbatches have been processed by backward computation block of this stage, the corresponding AllReduce operation is executed.

Algorithm 1 Pipeline Execution Scheduler - ***PE***

Input: $\mathcal{G}(\mathcal{V},\mathcal{E}), \mathcal{S}, \mathcal{F}: S \to \mathbb{P}(\mathcal{V})$
Output: T_{PE}, **e**

Initialize execution order queues $U_s \leftarrow \emptyset, \forall s \in \mathcal{S}$
Initialize available microbatch queues $Q_1 \leftarrow \{1, 2, \ldots, M\}$ and $Q_j \leftarrow \emptyset, \forall j \in \mathcal{J}/\{1\}$
while $\exists j \in \mathcal{J}, Q_j \neq \emptyset$ **do**
  for $j \in \mathcal{J} : Q_j \neq \emptyset$ **do**
    Pop one microbatch m out from the head of Q_j, and push m to the end of Q_{j+1} if $j < |\mathcal{J}|$
    Add (m, j) to the corresponding U_s if block j is a computation block
  end for
end while
Pop the first $(1, 1)$ out of U_{s_1}, and set $e^f_{1,s_1} = 0$
while $\exists s \in \mathcal{S}$, a block of stage s completes or a communication block which transmits data to stage s finishes at time t **do**
  if $s \in \mathcal{S}_{repl}$ and $U_s = \emptyset$ **then**
    Start AllReduce operation, and set e^A_s to t
  end if
  if $U_s = \emptyset, \forall s \in \mathcal{S}$ **then**
    break
  end if
  if a computation block of s finishes **then**
    Start successor communication block
  end if
  if the first (microbatch index, block number) in U_s is ready **then**
    Pop (microbatch index, block number) out of U_s
    Start the block and set the $e^f_{m,s}$ or $e^b_{m,s}$ to t
  end if
end while
Calculate the makespan: $T_{PE} = \max\{\max_{m\in\{1,2,\ldots,M\}}(e^b_{m,s_1} + \frac{\sum_{l\in s_1} p^b_l}{|\mathcal{F}(s_1)|}), \max_{s\in\mathcal{S}_{repl}}(e^A_s + A_s)\}$
Return T_{PE}, **e**

We summarize our pipeline execution scheduling algorithm in Alg. 1. The following lemma gives an upper bound of the per-iteration training time achieved by this PE algorithm.

Lemma 1. *Per-iteration training time achieved by Alg. 1, T_{PE}, is no larger than* $(1 + \frac{4|\mathcal{S}|-4}{M})M\mathcal{C} + \max_{s\in\mathcal{S}_{repl}}\{A_s\}$, *where* $\mathcal{C} = \max\{\max_{n\in\{1,\ldots,|\mathcal{S}|\}} \frac{\sum_{l\in s_n}(p^f_l + p^b_l)}{|\mathcal{F}(s_n)|}, \max_{n\in\{1,\ldots,|\mathcal{S}|-1\}}\{c^f_{s_n,s_{n+1}} + c^b_{s_{n+1},s_n}\}\}$, *denoting the maximum time to process a microbatch on a single stage (including both forward and backward computation) or an inter-stage communication channel (including data transfer in both forward and backward propagation phases), without considering AllReduce operations.*

We have all the missing proofs in our technical report.

B. DNN Partition and Device Mapping Algorithm

Lemma 1 shows that the per-iteration training time is positively related to the number of stage $|\mathcal{S}|$ that DNN $\mathcal{D}$ is partitioned into, the maximum time $\mathcal{C}$ to process a microbatch on a single stage or communication channel, and the maximum AllReduce operation time among replicated stages. The number of stage partitions, $|\mathcal{S}|$, is at most the same as the number of GPUs, V. We next design a model partition and device mapping algorithm aiming at minimizing the maximum time, $\mathcal{W}(|\mathcal{S}|)$, to process all microbatches on a single stage (including AllReduce operations) or a communication channel, given the number of stages $|\mathcal{S}|$. The purpose is to minimize the upper bound of per-iteration training time in Lemma 1, as $\mathcal{W}(|\mathcal{S}|)$ is related to both $\mathcal{C}$ and $\max_{s\in\mathcal{S}_{repl}}\{A_s\}$:

$$\mathcal{W}(|\mathcal{S}|) = \max\{ \max_{s\notin\mathcal{S}_{repl}}\{M\sum_{l\in s}(p^f_l + p^b_l)\}, \max_{s\in\mathcal{S}_{repl}}\{M\frac{\sum_{l\in s}(p^f_l+p^b_l)}{|\mathcal{F}(s)|} + A_s\}, \max_{n\in\{1,\ldots,|\mathcal{S}|-1\}}\{M(c^f_{s_n,s_{n+1}} + c^b_{s_{n+1},s_n})\}\}$$

Our DNN partition and device mapping problem, without considering stage replication, can be reduced to the NP-complete problem of pipeline partition over a heterogeneous communication platform [29], which partitions a workflow among a cluster of devices with heterogeneous connectivity to maximize the pipeline throughput. We design an efficient *balanced pipeline partition and device mapping algorithm* (BPPM) to derive a near-optimal solution, which includes two components: 1) a device ordering module that calculates a linear ordering of all GPUs; and 2) an algorithm that partitions the DNN model onto GPUs respecting the device order.

1) Recursive device ordering (RDO): We decide a linear ordering of GPUs in $\mathcal{V}$: $\{v_1, v_2, \ldots, v_V\}$. We will map stages (stage replicas) to devices according to this ordering and stage dependencies, *i.e.*, (replicas of) the first stage mapped to the device(s) at the head of the device ordering, and then the next stage to latter device(s), etc. We target a linear ordering with maximal bandwidth between consecutive GPUs, such that the bandwidth between stages and between replicas of the same stage is maximized. A recursive min-cut algorithm is designed to find the ordering in polynomial time, as given in Alg. 2. $rank_l$ ($rank_h$) represents the lowest (highest) rank of devices in the current subgraph in the ordering, initialized to 1 and V, respectively (in the complete Alg. 3 that invokes RDO). In each recursive step, we find a min-cut within the current input graph (leveraging an efficient min-cut algorithm in [30]), partition the graph into two subgraphs accordingly, and call RDO again to order devices in the two subgraphs, respectively. When the input graph contains only one GPU, we assign $rank_l$ (which equals $rank_h$) to it. We order GPUs according to their computed ranks in ascending order.

Algorithm 2 Recursive Device Ordering - ***RDO***

Input: $G(\mathcal{V},\mathcal{E}), rank_l, rank_h$
if $|\mathcal{V}| == 1$ **then**
  Set $rank(v) \leftarrow rank_l$
else
  $G^1(\mathcal{V}^1,\mathcal{E}^1), G^2(\mathcal{V}^2,\mathcal{E}^2) = \text{min-cut}(G)$
  RDO$(G^1, rank_l, rank_l + |\mathcal{V}^1| - 1)$
  RDO$(G^2, rank_l + |\mathcal{V}^1|, rank_h)$
end if

The link(s) within the min-cut will be used for inter-stage communication between the two consecutive stages mapped respectively onto two GPUs in the two subgraphs, or AllReduce operation within one stage whose replications are mapped into both subgraphs. Since all GPUs will be used in pipeline training, at least one link in each min-cut needs to be used for communication (as otherwise the training topology will not be a connected graph). By dividing the device graphs in this way, link(s) in each min-cut will be used only between two stages or among replicas of one replicated stage, but not between two pairs of consecutive stages or replicas of two replicated stages. Hence, links with small bandwidth are minimally exploited while large-bandwidth links are maximally used for inter-stage or AllReduce communication, minimizing the maximum communication time on a single communication channel.

2) Pipeline partition, replication and mapping (PRM): Following the device ordering $\{v_1, v_2, \ldots, v_V\}$, we leverage dynamic programming to partition and map $\mathcal{D}$ onto the GPUs that minimizes the maximum execution time $\mathcal{W}(|\mathcal{S}|)$ on a single stage or communication channel.

Let $W(l, \xi, r, i)$ denote the optimal (aka minimum) maximum execution time on a single GPU or communication channel, when we partition the first l consecutive layers in $\mathcal{D}$ into ξ stages with the last stage replicated into r replica(s) ($r \geq 1$), and place the stages over GPUs v_1 to v_i. We have $\mathcal{W}(|\mathcal{S}|) = \min_{1 \leq r \leq V} W(L, |\mathcal{S}|, r, V)$. $W(l, \xi, r, i)$ can be recursively computed as follows:

$$W(l,\xi,r,i) = \min_{1\leq l'\leq l-1, 1\leq r'\leq i-r} \max\{W(l',\xi-1,r',i-r),$$
$$M\frac{d^f_{l',l'+1}+d^b_{l',l'+1}}{r'rb_{r'r}}, M\frac{\sum_{o=l'+1}^{l}(p^f_o+p^b_o)}{r} + A_{l'+1\to l}(v_{i-r+1}\to v_i)\}$$

The first term inside $\max$ is the maximal execution time on a single GPU/communication channel by optimal partition of layers 1 to l' into $\xi - 1$ stages (with the last stage replicated on r' GPUs) and mapping them on GPUs v_1 to v_{i-r}. The second term computes the total communication time on the communication channel between layers l' and $l'+1$, where $b_{r'r} = \min_{v' \in \{v_{i-r-r'+1}, \ldots, v_{i-r}\}, v \in \{v_{i-r+1}, \ldots, v_i\}} b_{v',v}$ is the minimal link bandwidth between r' replicas of layer l' and r replicas of layer $l'+1$. The third term is the training time on the last stage, including processing time of all microbatches over layers $l'+1$ to l replicated on r GPUs and time taken by the corresponding AllReduce operation. Here $A_{l'+1\to l}(v_{i-r+1} \to v_i)$ denotes the time for AllReduce operation of layers $l'+1$ to l replicated over GPUs v_{i-r+1} to v_i.

To compute $W(l, \xi, r, i)$, we solve the subproblem of optimally partitioning the first l' layers into $\xi - 1$ stages on GPUs v_1 to v_{i-r}, while replicating the stage containing layers $l'+1$ to l over GPUs v_{i-r+1} to v_i. We consider all possible choices of layer l' and various replication strategies of the stage containing l', and decide $W(l, \xi, r, i)$ as the minimal time computed among them. The detailed dynamic programming PRM algorithm is omitted due to space limit, which takes $G(\mathcal{V}, \mathcal{E})$, $\{v_1, v_2, \ldots, v_V\}$, $\mathcal{D}$, L, V, ξ, r as input, and output $W(L, \xi, r, V)$ as well as $\mathcal{S}$ and $\mathcal{F}$. The following lemma shows that the best stage partition and device mapping that our algorithms identify when the given number of stages varies, achieves a maximum per-stage/communication channel execution time close to optimum.

Algorithm 3 Synchronous Pipeline Planning - ***SPP***

Input: $G(\mathcal{V}, \mathcal{E}), \mathcal{D}$
Output: $\bar{\mathcal{S}}, \bar{\mathcal{F}}, \bar{\mathbf{e}}, T_{SPP}$
1: ***RDO***$(\mathcal{G}(\mathcal{V}, \mathcal{E}), 1, V)$
2: Obtain device ordering $\{v_1, \ldots, v_V\}$ according to $rank(v), \forall v \in \mathcal{V}$
3: $T_{SPP} \leftarrow$ **INF**
4: **for** $\xi \in \{1, 2, \ldots, V\}$ **do**
5: **for** $r \in \{1, 2, \ldots, V\}$ **do**
6: $W(L, \xi, r, V), \mathcal{S}_r, \mathcal{F}_r \leftarrow$ ***PRM***$(G(\mathcal{V}, \mathcal{E}), \{v_1, \ldots, v_V\}, \mathcal{D}, L, V, \xi, r)$
7: **end for**
8: Set $\mathcal{S}$ and $\mathcal{F}$ to $\mathcal{S}_r$ and $\mathcal{F}_r$ that achieve the minimum $W(L, \xi, r, V)$
9: $T_{PE}, \mathbf{e} \leftarrow$ ***PE***$(\mathcal{G}, \mathcal{S}, \mathcal{F})$
10: $T_{SPP} \leftarrow T_{PE}, \bar{\mathcal{S}} \leftarrow \mathcal{S}, \bar{\mathcal{F}} \leftarrow \mathcal{F}, \bar{\mathbf{e}} \leftarrow \mathbf{e}$ if $T_{PE} < T_{SPP}$
11: **end for**
12: Return $\bar{\mathcal{S}}, \bar{\mathcal{F}}, \bar{\mathbf{e}}, T_{SPP}$

Lemma 2. *Let $\mathcal{W}_{PRM} = \min_{|\mathcal{S}| \in \{1,\ldots,V\}} \mathcal{W}(|\mathcal{S}|)$. $\mathcal{W}_{PRM}$ achieved by RDO and PRM is no larger than $(1 + \Phi)$ times $\mathcal{W}^*$, the optimal (aka minimum) maximum execution time on a single stage or communication channel, where $\Phi = \frac{\max\{p_{max} b_{max}, d_{max}\}}{\Gamma}(\frac{1}{b_{min}} - \frac{1}{b_{max}})$, $d_{max} = \max_{1 \leq l \leq L-1}(d^f_{l,l+1} + d^b_{l+1,l})$, $p_{max} = \max_{1 \leq l \leq L}(p^f_l + p^b_l)$, and $\Gamma = \sum_{1 \leq l \leq L}(p^f_l + p^b_l)/V$.*

C. Complete Synchronous Pipeline Planning Algorithm

Our complete synchronous pipeline planning (*SPP*) algorithm is given in Alg. 3, which produces model partition $\bar{\mathcal{S}}$, device mapping $\bar{\mathcal{F}}$ and execution schedule $\bar{\mathbf{e}}$. We first leverage RDO in Alg. 2 to obtain a linear ordering of all GPUs (lines 1-2). We next vary the number of stages from 1 to V (line 4): given a stage number to partition the model into, we vary r and call PRM to compute the best stage partition and device mapping (lines 5-8) that achieve $\mathcal{W}(|\mathcal{S}|)$; we then invoke PE in Alg. 1 to compute execution schedule of microbatches over these partitions on the respective devices (line 9). We identify the best stage partition number as the one minimizing the makespan of a training iteration (lines 10-12) together with the corresponding $\bar{\mathcal{S}}$, $\bar{\mathcal{F}}$ and $\bar{\mathbf{e}}$.

Theorem 1. *The makespan of a training iteration achieved by SPP, T_{SPP}, is less than $(2 + \frac{4V-4}{M})(1 + \Phi)$ times the optimal makespan, T^*.*

Theorem 2. *Our complete synchronous pipeline planning Alg. 3 runs in polynomial time.*

V. Performance Evaluation

We evaluate *SPP* with both testbed experiments and simulation studies.

A. Testbed experiments

Implementation. We implement *SPP* using C++ and Python on Tensorflow 1.14.1 [4]. We use Tensorflow profiler to collect runtime data of each layer of each DNN model

TABLE II: Benchmark DNN models

Model	# of parameters	# of microbatches	microbatch size (# of samples)
VGG19 [34]	144M	8	32
ResNet152 [2]	60M	4	4
Inception-V3 [35]	24M	8	32
Transformer [36]	55M	8	32
BERT-large [1]	340M	4	4
XLNet-large [37]	550M	4	4
BERT-48 [1]	640M	4	4 - 1080Ti×8 2 - V100×4

(*e.g.* forward/backward computation time, parameter size and activation size) over 20 training iterations. We assign a priority to each stage or AllReduce operation (implemented using NCCL collective AllReduce [31]) based on our computed execution order, such that they can be scheduled by TensorFlow execution engine accordingly.

Testbed. We evaluate *SPP* in two testbed environments: (1) One consists of 4 GPU servers, inter-connected by a Dell Z9100-ON switch, with 50Gbps peak bandwidth between any two servers. Each server has one 8-core Intel E5-1660 CPU, two GTX 1080Ti GPUs and one 50GbE NIC. (2) The other is a single server equipped with 4 Tesla V100 GPUs, two 10-core Intel Xeon E5-2630 v4 CPUs and a 100GbE NIC. GPUs in the sever are connected with 128Gbps PCIe bus.

DNN models. We train 7 representative DNN models: three image classification models on the ImageNet dataset [32] and four NLP models on SQuAD2.0 dataset [33] (Table II). The number of microbatches and microbatch size for training each model are set as the maximum number×size (aka mini-batch size) without causing OOM (out of memory) for most baselines. The large batch sizes we use are consistent with common practice [16]. To run *SPP*, we modified ResNet152 by ignoring shortcut connections and Inception-V3 by aggregating parallel branches (branches with the same start point and end point) as one layer. We apply *SPP* to the modified models to decide the strategies, and then train the original models (without the modifications) using the obtained strategies.

Baselines. *SPP* is compared with 4 state-of-the-art schemes: (i) Data Parallelism (DP), with each GPU training the complete model with $\frac{\text{mini-batch size}}{\text{\# of GPUs}}$ amount of data; (ii) GPipe [15]; (iii) PipeDream [10]; (iv) HetPipe [12] (see Sec. II for details of the latter three). Unless stated otherwise, we enforce a synchronization barrier at the end of each training iteration in PipeDream and HetPipe, removing the negative impact of asynchronous training on model convergence.

Per-iteration training speed-up. We compare *SPP* with all baselines in terms of per-iteration training time in the first testbed environment (1080Ti×8). In the other environment (V100×4), we omit HetPipe as all four GPUs are on the same server, reducing HetPipe to PipeDream solutions. In Table III, the speed-up is computed by $\frac{\text{Baseline time}-\text{SPP time}}{\text{SPP time}}$. *SPP* outperforms the baselines in all cases. While both DP and HetPipe require an AllReduce operation to synchronize gradients, *SPP* incurs less parameter synchronization traffic and maximally overlaps communication with computation within each training iteration. As a result, *SPP* outperforms them by more than 20% in most cases. The large speed-up for *SPP* over baselines on Inception-V3 demonstrates that our design handles a model with non-uniform layer computation time well. As VGG-19 has a small number of layers that can be easily optimally partitioned, we observe minor gain comparing *SPP* with PipeDream and GPipe. For BERT-large on 1080Ti×8 testbed, PipeDream partitions the model into uniform stages, achieving similar good performance as *SPP*.

End-to-end training performance. We next compare training convergence among *SPP*, DP, GPipe and PipeDream. For PipeDream, we use its original asynchronous pipeline design. Fig. 4 shows the training progress of VGG19 on the V100×4 testbed to achieve a target 90% top-5 accuracy [32]. *SPP* achieves the target accuracy using the least training time. Despite only marginal speed-up in per-iteration training time as compared to PipeDream (using its synchronous pipeline mode that we implemented), here *SPP* outperforms PipeDream (with original asynchronous pipeline design) by 9.05% in terms of the end-to-end training time. This is because PipeDream's asynchronous pipeline training slows down the model convergence progress, as training microbatches on outdated versions of model parameters [14].

B. Trace-driven Simulation

Settings. By default, we simulate training of BERT-large (27-layers including 24 transformer layers) with 32 microbatches and a microbatch size of 6 on 8 servers, each equipped with 4 GPUs. We drive our simulation using profiled data collected by running the DNN on a V100 GPU. 3 servers have intra-server bandwidth between [96, 128] Gbps (representing PCIe links [38]), and the other 5 servers [160, 200] Gbps (representing NVLink connections [39]). By default, inter-server bandwidth is set within [32, 40] Gbps to emulate an RDMA network [40].

1) Different numbers of microbatches: Fig. 5 shows that *SPP* achieves significant training speed-up compared with the four baselines at different microbatch numbers (M). We also observed (figure omitted due to space limit) that higher GPU utilization is achieved with *SPP* when M is larger, implying the diminishing gap between *SPP* and the optimal solution (which maximally utilizes GPUs for the best training speed). This is consistent with Theorem 1: as M increases, the approximation ratio becomes smaller (*i.e.*, better).

2) Different inter-server bandwidth levels: We emulate three types of inter-server networks with low, medium and high bandwidth, respectively. Fig. 6 shows that per-iteration training time of *SPP*, GPipe and PipeDream is stable at different bandwidth levels, while performance of DP and HetPipe drops dramatically at small bandwidth. This is because the former three overlap most inter-stage communication and AllReduce operations with computation, achieving higher GPU utilization even with small communication bandwidth. DP and HetPipe require an AllReduce operation over inter-server connections at the end of each training iteration, which incurs large communication time with small bandwidth.

3) Different inter-GPU connectivity: We next vary the number of available GPUs on servers and the server number.

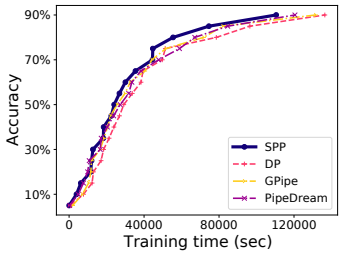


Fig. 4: VGG19 training progress: *SPP* vs. baselines

Model	Testbed	*SPP*	DP (Speed-up)	GPipe (Speed-up)	PipeDream (Speed-up)	HetPipe (Speed-up)
VGG19	1080Ti×8	1.799	2.882 (60.2%)	2.120 (17.9%)	1.949 (8.3%)	2.696 (49.9%)
	V100×4	0.983	1.245 (26.7%)	1.004 (2.1%)	1.024 (4.2%)	-
ResNet152	1080Ti×8	0.732	0.896 (22.4%)	1.214 (65.8%)	OOM (-)	0.843 (15.2%)
	V100×4	0.832	1.209 (45.3%)	1.041 (25.1%)	0.873 (4.9%)	-
Inception-V3	1080Ti×8	0.303	0.420 (38.6%)	0.551 (81.8%)	0.656 (116.5%)	0.408 (34.7%)
	V100×4	0.357	0.663 (85.7%)	0.587 (64.4%)	0.919 (157.4%)	-
Transformer	1080Ti×8	0.640	0.944 (47.5%)	1.234 (92.8%)	1.118 (74.7%)	0.766 (19.7%)
	V100×4	1.065	2.533 (137.8%)	1.487 (39.6%)	1.830 (71.8%)	-
BERT-large	1080Ti×8	0.409	0.524 (28.1%)	0.472 (15.4%)	0.421 (2.9%)	0.525 (28.4%)
	V100×4	0.952	2.269 (138.3%)	1.665 (74.9%)	1.084 (13.9%)	-
XLNet-large	1080Ti×8	1.299	1.388 (6.7%)	1.696 (30.6%)	1.384 (6.5%)	1.628 (25.3%)
	V100×4	1.437	1.842 (28.3%)	1.720 (19.7%)	1.690 (17.6%)	-
BERT-48	1080Ti×8	0.762	OOM (-)	1.885 (147.4%)	1.266 (66.1%)	1.377 (80.7%)
	V100×4	0.855	1.656 (93.7%)	1.199 (40.2%)	1.160 (35.7%)	-

TABLE III: Per-iteration training time (in seconds) of different DNN models

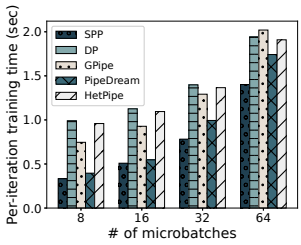


Fig. 5: Training time: different # of microbatches

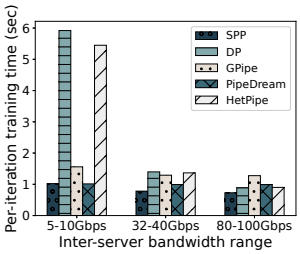


Fig. 6: Training time: different inter-server bandwidth levels

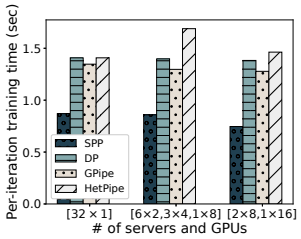


Fig. 7: Training time: different inter-GPU connectivity

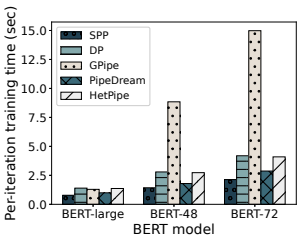


Fig. 8: Training time: BERT with different # of layers

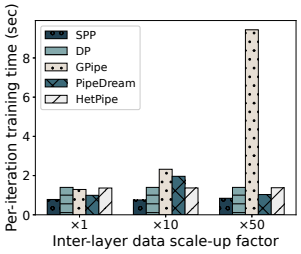


Fig. 9: Training time: different inter-layer data sizes

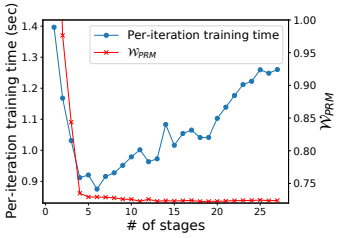


Fig. 10: Training time & $\mathcal{W}_{PRM}$: different # of stages

In Fig. 7, $[6 \times 2, 3 \times 4, 1 \times 8]$ represents training the model over 6 servers each with 2 GPUs, three 4-GPU servers and one 8-GPU server. *SPP* achieves the best performance in all inter-GPU connection topologies.

4) Different numbers of layers: Fig. 8 compares the training performance of BERT-large, BERT-48 (48 Transformer layers) and BERT-72 (72 Transformer layers). As the model size increases, it is more difficult to obtain optimal model partition and device mapping solution. However, performance of *SPP* and PipeDream remains quite stable, with *SPP* outperforming PipeDream by more than 20% on the three models.

5) Different inter-layer data sizes: We investigate the impact of activation sizes which influence inter-stage communication time, by scaling the activation data in BERT-large by different factors. Fig. 9 shows that the per-iteration training time with *SPP* remains similar with the increase of activation sizes, due to the excellent communication and computation overlap it achieves. GPipe tends to partition the model into more stages, resulting in more inter-stage communication time.

6) Different numbers of stages: Lemma 1 gives that the performance of our algorithm is related to: (1) the number of stages $|\mathcal{S}|$, and (2) $\mathcal{W}_{PRM}$, the maximum time to process all microbatches on a single stage or communication channel. While PipeDream only aims at minimizing $\mathcal{W}_{PRM}$, *SPP* strikes a balance between the two factors. In Fig. 10, $\mathcal{W}_{PRM}$ first decreases when the model is partitioned into more stages, and becomes stable starting from the stage number of 4. The main reason is that for training BERT-large (with 24 uniform Transformer layers) on 32 GPUs, the per-stage training time with 4 stages is already quite close to the optimal per-stage training time.[3] The training time first decreases as $\mathcal{W}_{PRM}$ drops, and then increases when $\mathcal{W}_{PRM}$ stabilizes and $|\mathcal{S}|$ becomes the dominant factor, which is consistent with Lemma 1. This indicates that only minimizing $\mathcal{W}_{PRM}$ does not yield the best solution. *SPP* strategically selects the 6-stage partition solution to minimize per-iteration training time.

VI. Conclusion

This paper designs efficient algorithms for expediting synchronous pipeline training of DNNs over arbitrary inter-GPU connectivity. We partition a given DNN, replicate and distribute the partitions over available GPUs, and design an efficient scheduler to order pipeline execution of microbatches over partitioned stages on different GPUs, minimizing the training time. Our comparative experiments on two GPU testbeds prove that our design outperforms state-of-the-art approaches up to 157%. Trace-driven simulations further show our algorithms' superiority under various settings.

[3]With 4 stages, we roughly have 6 layers per stage and each stage replicated to 8 GPUs; per-stage time is 6p/8 (p denotes per-layer computation time) plus AllReduce time. Optimal per-stage training time is lower bounded by 24p/32.

REFERENCES

[1] 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.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. of IEEE CVPR*, 2016.

[3] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots That Can Adapt like Animals," *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *Proc. of USENIX OSDI*, 2016.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proc. of NeurIPS*, 2019, pp. 8024–8035.

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *Proc. of USENIX OSDI*, 2014.

[7] A. Sergeev and M. Del Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[8] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using GPU Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[9] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the Straggler Problem for Iterative Convergent Parallel ML," in *Proc. of ACM SoCC*, 2016.

[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 *Proc. of ACM SOSP*, 2019.

[11] J. Geng, D. Li, and S. Wang, "ElasticPipe: An Efficient and Dynamic Model-Parallel Solution to Dnn Training," in *Proc. of the 10th Workshop on Scientific Cloud Computing*, 2019.

[12] J. H. Park, G. Yun, C. M. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y. ri Choi, "HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism," in *Proc. of USENIX ATC*, 2020.

[13] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-Efficient Pipeline-Parallel DNN Training," in *Proc. of ICML*, 2021.

[14] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *Proc. of NeurIPS*, 2013.

[15] 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," in *Proc. of NeurIPS*, 2019.

[16] 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 *Proc. of ACM PPoPP*, 2021, pp. 431–445.

[17] *NVIDIA DGX-1*, https://www.nvidia.com/en-us/data-center/dgx-1/.

[18] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-aware MPI on RDMA-enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, 2013.

[19] L. A. Goldberg, M. Paterson, A. Srinivasan, and E. Sweedyk, "Better Approximation Guarantees for Job-Shop Scheduling," *SIAM Journal on Discrete Mathematics*, vol. 14, no. 1, pp. 67–92, 2001.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016.

[21] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device Placement Optimization with Reinforcement Learning," in *Proc. of ICML*. PMLR, 2017, pp. 2430–2439.

[22] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, "Placeto: Efficient Progressive Device Placement Optimization," in *NIPS Machine Learning for Systems Workshop*, 2018.

[23] X. Yi, S. Zhang, Z. Luo, G. Long, L. Diao, C. Wu, Z. Zheng, J. Yang, and W. Lin, "Optimizing Distributed Training Deployment in Heterogeneous GPU Clusters," in *Proc. of International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 93–107.

[24] X. Wu, H. Xu, B. Li, and Y. Xiong, "Stanza: Layer Separation for Distributed Training in Deep Learning," *IEEE Transactions on Services Computing*, pp. 1–1, 2020.

[25] X. Yi, Z. Luo, C. Meng, M. Wang, G. Long, C. Wu, J. Yang, and W. Lin, "Fast Training of Deep Learning Models over Multiple GPUs," in *Proc. of the 21st International Middleware Conference*, 2020, pp. 105–118.

[26] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, "Efficient and Robust Parallel DNN Training Through Model Parallelism on Multi-GPU Platform," *arXiv preprint arXiv:1809.02839*, 2018.

[27] I. Colin, L. Dos Santos, and K. Scaman, "Theoretical Limits of Pipeline Parallel Optimization and Application to Distributed Deep Learning," in *Proc. of NeurIPS*, 2019.

[28] *Performance reported by NCCL tests*, https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md.

[29] A. Benoit and Y. Robert, "Mapping Pipeline Skeletons onto Heterogeneous Platforms," *Journal of Parallel and Distributed Computing*, vol. 68, no. 6, pp. 790–808, 2008.

[30] M. Stoer and F. Wagner, "A Simple Min-Cut Algorithm," *Journal of the ACM*, vol. 44, no. 4, pp. 585–591, 1997.

[31] *NVIDIA Collective Communication Library*, https://github.com/NVIDIA/nccl.

[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proc. of IEEE CVPR*, 2009.

[33] P. Rajpurkar, R. Jia, and P. Liang, "Know What You Don't Know: Unanswerable Questions for SQuAD," *arXiv preprint arXiv:1806.03822*, 2018.

[34] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[35] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proc. of IEEE CVPR*, 2016.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Proc. of NeurIPS*, 2017.

[37] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized Autoregressive Pretraining for Language Understanding," in *Proc. of NeurIPS*, 2019.

[38] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie, "Evaluating on-Node GPU Interconnects for Deep Learning Workloads," in *Proc. of International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2017.

[39] M. Amaral, J. Polo, D. Carrera, S. Seelam, and M. Steinder, "Topology-Aware GPU Scheduling for Learning Workloads in Cloud Environments," in *Proc. of SC*, 2017.

[40] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda, "Multi-Path Transport for RDMA in Datacenters," in *Proc. of USENIX NSDI*, 2018.