Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism

Abstract

Fine-tuning is an increasingly common technique that leverages transfer learning to dramatically expedite the training of huge, high-quality models. Critically, fine-tuning holds the potential to make giant state-of-the-art models pre-trained on high-end super-computing-grade systems readily available for users that lack access to such costly resources. Unfortunately, this potential is still difficult to realize because the models often do not fit in the memory of a single commodity GPU, making fine-tuning a challenging problem.

We present FTPipe, a system that explores a previously unexplored dimension of pipeline model parallelism, making multi-GPU execution of fine-tuning tasks for giant neural networks readily accessible. A key novel concept, called Mixed-pipe, allows balancing the compute- and memory-load on the GPUs by partitioning the model into computational blocks of any granularity while relaxing model topology constraints. Our system goes beyond topology limitations of previous pipeline-parallel approaches, efficiently training a new family of models, including the current state-of-the-art. Our extensive experiments on giant NLP models (BERT-340M, GPT2-1.5B, and T5-3B) show that FTPipe achieves up to 3× speedup and state-of-the-art accuracy when fine-tuning giant transformers with billions of parameters. These models require from 12GB to 59GB of GPU memory, and FTPipe executes them on 8 commodity RTX2080-Ti GPUs, each with 11GB memory and standard PCIe. FTPipe will be publicly released on GitHub.

1 Introduction

Fine-tuning deep neural network (DNN) models is a technique commonly used to achieve state-of-the-art model quality on a wide range of tasks, such as question answering, text generation, translation, and more [42, 50, 51]. In fine-tuning, the model is not trained from scratch. Instead, a short training phase is performed on an already existing model, which has been pre-trained on large application-related datasets to obtain general domain knowledge [14]. By training on user-specific datasets, fine-tuning allows accommodating new inputs [10, 40, 55].

Fine-tuning is becoming increasingly important as models grow to billions of parameters (Figure 1). Since training such *giant* models from scratch is practical only on super-computer-scale systems [44, 46, 55], a provider-scale pre-training, followed by short fine-tuning on user data, is the key to making the power of giant models broadly accessible [52].

Unfortunately, fine-tuning giant models on *low-cost commodity hardware* is still a significant challenge. Small memory sizes and slow inter-GPU communications are the main characteristics that set apart commodity GPUs from high-end devices, and they dramatically affect the training of giant models.

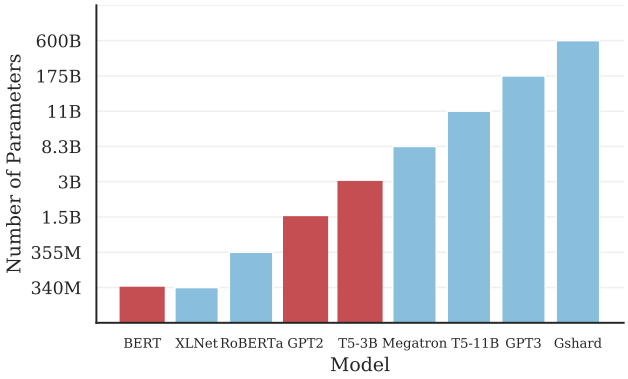

Figure 1: Model size trends. None of these models can be trained on a single NVIDIA RTX-2080-Ti commodity GPU due to memory requirements. Bars in red represent the models evaluated in this paper. FTPipe automatically transforms their sequential implementation into a multi-GPU one.

Small GPU memory is the key problem. For example, about 40 GBs are needed to train T5 model with 3B parameters for Recognizing Textual Entailment (RTE) [51] even with the smallest batch size of 1. This is larger than the memory size of any commodity GPU.

Existing approaches to cope with GPU memory constraints, such as sharding/swapping [41, 49], are ineffective due to slow PCIe communications. This problem is further exacerbated in fine-tuning tasks because the typically-small batch sizes do not allow for overhead mitigation using communication and computation overlapping. Therefore, using a single GPU, or a data-parallel multi-GPU approach for fine-tuning giant models are not viable options.

Model-parallel techniques allow fitting the model into memories of multiple GPUs. However, for commodity GPUs, the poor communication performance rules out the most commonly used Intra-Layer Model Parallelism approach, where each DNN layer is split among multiple GPUs. High volumes of all-to-all communications (AllReduce) with many synchronization points [44, 46] perform poorly over PCIe.

Pipeline-parallel execution exploits the depth dimension of DNN models [15, 33], distributing them across GPUs at a coarse granularity, i.e., DNN layers. A model is partitioned into multiple consecutive stages according to its topology.

Each stage runs on a different GPU. During training, the input samples are streamed through the pipeline. This approach allows training giant models on multiple GPUs without the high communication costs of intra-layer parallelism, making it ideally suitable for commodity hardware.

However, existing methods have some drawbacks when applied to fine-tuning giant models.

GPipe [15] demonstrated the benefits of pipeline parallelization, but has pipeline bubbles, leading to low hardware utilization for small mini-batches typical for fine-tuning tasks [50]. The bubbles are caused by GPipe's *synchronous* approach whereby the next mini-batch of samples does not start before the previous one traverses the whole pipeline.

PipeDream [33] improved GPU utilization by introducing *asynchronous* training, where the mini-batches no longer wait for each other to complete. However, PipeDream stores multiple versions of the weights in GPU memory (*weight stashing*) to mitigate the effects of *weight staleness* [56] inherent to asynchronous training, dramatically reduces the size of the models that can be trained [1].

Last, both GPipe and PipeDream may suffer from *poor load-balance* across GPUs in DNNs with a complex topology, such as giant encoder-decoder DNNs in Natural Language Processing (NLP). NLP features the largest models to which fine-tuning is most commonly applied, so enabling efficient fine-tuning for such models is of high practical importance.

We introduce **FTPipe**, a novel memory-efficient execution framework for pipelined fine-tuning of giant DNNs on commodity GPUs. Using FTPipe is easy: it *automatically* generates a functional performance-optimized pipeline-parallel multi-GPU version of a given sequential model implemented in PyTorch [35]. It enables the execution of models whose description does not adhere to the conventional sequences of coarse-grain layers.

FTPipe builds on two key observations:

Partitioning without DNN topology constraints. FTPipe optimizes the load balance by relaxing the model topology constraints when distributing it across the GPUs. Specifically, it introduces a novel *mixed-pipe* partitioning scheme which permits assigning any fine-grained combination of model operations to run on GPUs, *even when the assignment does not follow the original sequence of network-architecture layers.* This is in contrast to all prior approaches [15, 33] that use coarse-grain layer-granular partitioning, with only adjacent layers scheduled to run together on the same GPU. The mixed-pipe scheme exploits the advantages of a new, previously unexplored, point in the trade-off between inter-GPU communications and load-balancing, prioritizing the latter. It enables efficient pipeline-parallel training of neural networks common in NLP, including language models with shared embedding ("tied weights") [18,38] and giant Transformer-based encoder-decoder networks [48].

[1] PipeDream still has staleness, with parameter consistency similar to ASGD [9], see Table 1

Fine-tuning large models is less sensitive to staleness. FTPipe fastest version employs an asynchronous training scheme which is prone to staleness [3, 4]. However, large models with pre-trained weights have properties that make training less sensitive to staleness. These include smooth and slowly-varying optimization trajectories [26], or small (and diminishing) learning rates. Thus, we achieve state-of-the-art results while avoiding costly staleness mitigation techniques.

We evaluate FTPipe on challenging fine-tuning tasks with three modern giant NLP models: T5, GPT2, and BERT with 3B, 1.5B, and 340M parameters, respectively. These models cannot be trained on a GPU with 11GB memory. Among them, T5, not previously trained or fine-tuned on an asynchronous model-parallel system, has been particularly challenging due to the diverse computational requirements of different layers.

Our experiments on 8 NVIDIA RTX2080 Ti GPUs demonstrate significant performance benefits of FTPipe over GPipe, which is the only state-of-the-art pipeline execution framework that was able to train all the evaluated models on our hardware. For example, on T5 FTPipe is from 30% to 3× faster, with the same accuracy, of which Mixed-pipe gains 1.3×-1.82×, and asynchronous training an additional 1.19×-2.34× (results vary between datasets). This is the first execution of T5-3B on a model-parallel system we know of.

Compared to PipeDream, FTPipe trains significantly larger models and avoids partitioning solutions that do not fit in GPU memory. Further, FTPipe outperformed PipeDream by 13% when training BERT, after augmenting PipeDream implementation with checkpointing [6, 11].

Essentially, FTPipe generalizes the pipeline execution of former approaches to *dataflow-aware execution*, avoiding unnecessary communications and allowing more parallelism. Specifically, FTPipe stages can communicate with non-neighbors directly, without passing via adjacent stages. Thus, data is automatically loaded exactly where needed, and the stages can be concurrent in case the network graph allows it. As a result, model parameters and their gradients can be communicated thus making the execution of models with shared weights possible in pipelines.

In summary, our main contributions are as follows:

FTPipe. We present FTPipe, an automatic framework that transforms a sequential DNN implementation into a multi-GPU pipeline-parallel one, enabling fine-tuning of giant state-of-the-art neural networks on commodity GPUs,

Mixed-pipe. We present the fine-grain partitioning scheme which relaxes the model topology constraints when scheduling DNN computations on GPUs, thereby vastly improving load balance across the workers without additional communication overheads,

Evaluation. We evaluate FTPipe on challenging fine-tuning benchmarks, and giant transformers: GPT2 [39], BERT [10], and T5-3B [40], consistently outperforming GPipe and PipeDream while attaining state-of-the-art accuracy.

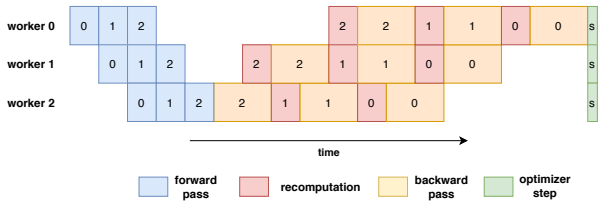

Figure 2: GPipe [15] synchronous training. Each rectangle (number is the micro-batch being processed) represents a pipeline stage running a certain task denoted by its color, except for the last optimizer step.

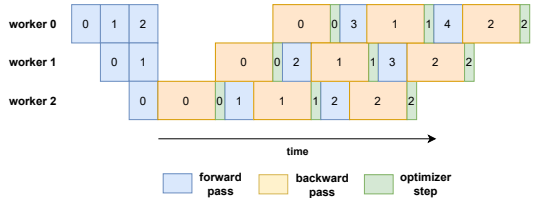

Figure 3: PipeDream [33] asynchronous training.

2 Background

Giant models. We call *giant* those models which do not fit into the memory of a single accelerator during training.

Pipelined training. A pipeline is trained by computing gradients for each *mini-batch*, i.e., a subset of examples assigned together for gradient computations. A mini-batch can be further sliced into smaller *micro-batches* to fit into workers' memory. Micro-batches also help parallelize the mini-batch computation. For example, in GPipe, micro-batches are streamed through the pipeline one-by-one (Figure 2). GPipe attempts to parallelize execution by having a micro-batch proceed to the next pipeline stage while the next micro-batch enters the previous stage (aka intra-batch parallelism). However, GPipe *synchronously* updates the model parameters at the end of the mini-batch execution using the gradients accumulated from all micro-batches, a technique called *gradient accumulation*.

Staleness. An *asynchronous pipeline* as in PipeDream [33] (Figure 3) begins the forward pass of the next mini-batch without waiting for the previous one to finish (aka inter-batch parallelism). In an asynchronous pipeline, the next mini-batch might use old parameters in its forward pass and newly updated parameters on its backward pass, or old parameters in both passes. Pipeline with more stages may have more mini-batches executing concurrently, therefore a higher difference between old and new parameters. This problem, known as *weight staleness* (or simply staleness), was shown to introduce significant disturbances to the training process, deteriorating final model accuracy and, in extreme cases, even preventing convergence altogether [8, 57]. Several methods were proposed to mitigate staleness [3, 56].

3 Motivation

3.1 Importance of fine-tuning giant models

DNN models are constantly growing to achieve higher quality. This trend is particularly pronounced in Neural Language Processing models (Figure 1), where larger models achieve significantly better results [5, 15, 24]. However, such giant models are increasingly hard to train without access to supercomputer-scale resources. For example, training XLNet [55] of 340M parameters using a data set of 158GB required 5.5 days on 512 Google TPUs [20]. Therefore, many pre-trained models have been made publicly available to allow their use by those who do not have access to such computing capabilities (e.g., NVIDIA Model catalog [1]).

From user point of view it is often desirable to further improve the pre-trained model and tailor it to a user-specific data set. The process, called *fine-tuning*, was shown to be effective across a wide range of tasks [14]. Fine-tuning operates on relatively small data sets and requires much less computing power to complete, compared to training a model from scratch.

Unfortunately, fine-tuning of giant models still require the whole model to be resident in GPU memory, which is a challenge for commodity GPUs. Attempts to fine-tune models by updating a subset of parameters [2, 13, 37] are not generally applicable and their results are usually worse than fine-tuning the entire model [40].

In summary, *fine-tuning of giant models poses a challenge that impedes their broader adoption.*

3.2 I/O benefits of pipeline model parallelism

Pipeline execution has the potential to fully overlap communication and computation in each pipeline stage, thus making the communication overhead negligible even on commodity hardware. There are two primary reasons:

Low communication volume. First, in pipeline-parallel training, as in any model-parallel training, GPUs communicate intermediate activations and activation-gradients, which, according to our experiments, are orders of magnitude smaller than parameter-gradients communicated in data-parallelism, and smaller than the number of state shards between workers. For example, a T5-3B model partitioned into 8 stages communicates *a total* of 456.4MB for each forward and backward pass across the whole pipeline (micro-batch size of 4, full precision). In comparison, in a data-parallel approach, each update requires collecting and aggregating 12GB of gradients *per worker*.

Overlapping computation and communication . Pipelines can overlap the communication with the next forward and/or backward passes, and sometimes also with the parameter update operation. This means that stages with large enough computation-to-communication ratio (for GPipe: ≥ 0.5) can

completely overlap communication and computation. This is indeed the case for all giant models we consider in this work. For example, the aforementioned T5-3B achieves a per-worker average computation-to-communication ratio of 0.96 and 0.98 for the forward and backward passes respectively, even when communicated over PCIe.

3.3 Challenging load balance

GPipe and PipeDream obey model topology constraints when distributing the model across GPUs. Specifically, only adjacent layers are scheduled to run together on the same GPU. Such an approach, which we call sequential pipeline, or *seq-pipe*, fails to balance the DNN partitions across workers for some models.

One specific example of practical importance is the encoder-decoder architecture, widely used in NLP for sequence-to-sequence tasks [7, 25, 40, 48]. Consider the T5 Text-To-Text Transformer model, which is currently the state-of-the-art for many NLP tasks. When trained for question answering, the question is fed into the encoder, and the answer is fed into the decoder. In many cases, the answers are much shorter than the questions (e.g., yes/no questions). Thus, such training inputs cause a compute imbalance between encoders and decoders, since the computations in attention layers scale quadratically with the input sequence length. Furthermore, the number of parameters in both layer types is equally large, so grouping many "lightweight" layers is not possible because it would exceed GPU memory. Thus, partitioning such models into a pipeline under strict model topology constraints (only adjacent layers allocated to the same GPU) is prone to severe compute and memory imbalance as illustrated in Figure 4.

PipeDream combines both pipeline- and data-parallelism, potentially solving the load imbalance by running compute-heavy stages on multiple GPUs using data parallel techniques. However, the amount of additional GPUs required may be impractical: consider the case of unbalanced encoder-decoders as in Figure 4, with N decoders, N encoders, and a computation-load ratio of k between encoder and decoder. A memory limit which allows placing no more than M encoders or decoders in the same device will imply that $\#GPUs = \frac{N}{M} \cdot K$ additional data parallel GPUs are required. In T5-3B for example, $M \leq 12$, $N = 24$, and K varies according to input sequence lengths of encoder and decoder (we measured $K = 5.5$ for two of our datasets where sequences where 512 and 4 respectively).

In FTPipe, we overcome the limitations of existing pipeline partitioning approaches and thus enable fast execution with state-of-the-art accuracy for challenging fine-tuning tasks on commodity GPUs.

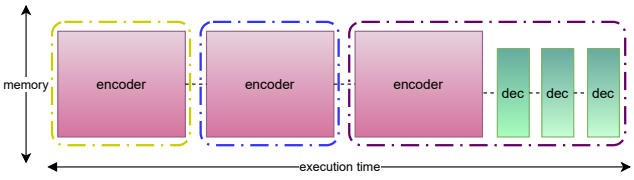

Figure 4: Partitioning unbalanced Encoder-Decoder models. Seq-pipes can assign only adjacent layers per GPU, thus are unable to partition the model for 3 GPUs in a balanced way (dashed lines). In contrast, Mixed-pipe can map one encoder and one decoder per GPU by assigning non-sequential layers to run together on the same GPU.

4 FTPipe

Overview. FTPipe is a system for training giant models with limited resources. It automatically transforms models which do not fit into the memory of a single accelerator into a multi-GPU pipelined training construction, and runs them on our data-flow execution runtime. A computational graph of the DNN model is decomposed into topologically sorted sub-graphs, grouped into pipeline stages according to their graph distance from the output (called *depth*). The number of stages is determined according to the GPU assignment as discussed in detail below. Computations are then performed according to a work scheduler.

We now describe individual steps.

Tracing execution. Given a model and inputs, FTPipe first identifies the directed acyclic computational data-flow graph of the network. A *computational graph representation* of a neural network is a directed graph $G(V,E)$ where each intermediate computational operation is a node $v \in V$ and each edge $(u,v) = e \in E$ represents a connection according to computation order which is determined by the forward pass (with an indication of whether its also being used for the backward pass). We refer to indivisible blocks of execution as *basic blocks*. By default, the smallest basic block is a single tensor operation (we do not split the underlying built-in compiled CUDA kernels).

Profiling. Each basic block is profiled to determine its memory consumption and execution time for each of its computational tasks, i.e., forward pass, recomputation (see Section 4.3) and backward pass. Aggregation of these values determines the block's memory and computing requirements used in the block-to-GPU assignment step. Tensor sizes for activations and gradients are also recorded, and used to compute communication times, given a bandwidth parameter.

We note that tracing and profiling pose an engineering challenge on its own, since the models we discuss cannot run on a single device, hence, for example, cannot be profiled nor executed with dummy inputs to trace their execution graph. We solve this problem by decoupling execution and swapping

layers, parameters, gradients, and activations in and out of host memory when needed.

Model partitioning. The objective of pipeline partitioning is to maximize throughput under memory and resource constraints. Maximizing throughput can also be described as minimizing the maximal *stage period* T_{max} among all the pipeline stages. We define T_i for stage i as

$$T_i = C + \max(0, T_{comm_{fwd}} - C) + \max(0, T_{comm_{bwd}} - C), \quad (1)$$

where $T_{comm_{fwd}}, T_{comm_{bwd}}, T_{compute_{fwd}}, T_{compute_{bwd}}$ are the time of communication and computations for forward and backward passes respectively (recomputation time is included in $T_{compute_{bwd}}$).

GPU assignment. The Mixed-pipe approach significantly inflates the search space of potential GPU allocations compared to Seq-pipe, to the point that - for the giant networks considered - an optimal, exhaustive assignment search (e.g., PipeDream's) is infeasible. Thus, FTPipe's GPU assignment takes a practical approach, combining efficient general graph partitioning and domain knowledge.

FTPipe employs several partitioning methods and let them compete, eventually selecting the best result according to its final throughput analysis via simulation. Among these schemes are (a) Mixed-pipe partitioning which searches mixed-pipe solutions as described in detail below, (b) PipeDream [33], which performs exhaustive search on the space of all Seq-pipes. (c) Acyclic [31,32] which performs a greedy search for Seq-pipes, for which we changed the objective to optimize pipeline throughput, and combined with memory constraints. It is useful when an exhaustive search is too long to execute (for a 2000 nodes graph exhaustive search on Seq-pipes takes around 20 minutes). (d) Metis [21], a general graph partitioning scheme that optimizes communications under load balancing constraints. The output can be either a Seq-pipe or a Mixed-pipe. It does not optimize pipeline throughput directly.

Work scheduling. While FTPipe supports both synchronous and asynchronous pipeline work schedules, full fine-tuning acceleration benefits are obtained using an asynchronous work scheduler illustrated at Figure 5. FTPipe is a general pipelined data-flow execution. It supports concurrent stages, shared weights, and data staging. FTPipe's work scheduling uses checkpointing with careful stage profiles (elaborated on in §4.3). In an asynchronous mode, for a stage of depth d, FTPipe first computes $d+1$ forward passes for different micro-batches, filling the pipeline, then moves into alternating between backward and forward passes, modifying the model parameters after each mini-batch.

We next discuss Mixed-pipe partitioning and GPU assignment in detail.

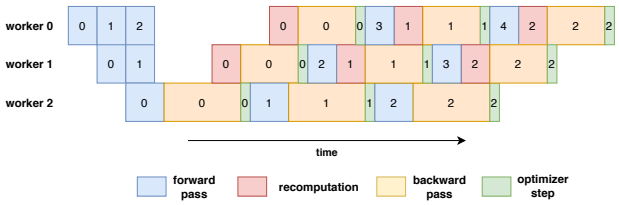

Figure 5: FTPipe asynchronous training. FTPipe uses checkpointing and recomputations to avoid storing multiple activations in memory. The last worker in the pipeline does not recompute, yet has the same period, load-balanced through profiling §4.3.

4.1 Mixed-pipe model partitioning

Seq-pipe partitioning proposed in prior works assigns only adjacent DNN layers to every GPU in the pipeline. As a result, such partitioning *implicitly* optimizes for reduced communications across GPUs, at the expense of fewer opportunities to balance the load among them. Our intuition for Mixed-pipe is that for giant models and commodity hardware, keeping the computations balanced is sometimes more important than reducing inter-GPU communications. Indeed, for giant models, the overhead of pipeline data transfers is relatively small compared to the computation load per accelerator (See §3). We therefore posit that relaxing the DNN topology constraints and allowing smaller, multiple partitions per GPU may improve load balance, while still affording communication-computation overlap.

However, there are three main challenges in doing so: (a) it is possible that small partitions might increase both communications and GPU invocation overheads; (b) without topology constraints, an exhaustive search for optimal partitioning [33] is not feasible; (c) for asynchronous pipelines, higher-depth partitions may imply additional staleness, which may harm model quality. The Mixed-pipe partitioning scheme addresses these issues, and our evaluation empirically shows that the final model quality is maintained.

For efficient partitioning that mixes blocks of operations among the GPUs, it is imperative that changing the assignment of a block from one GPU to another does not increase communication penalty. To achieve this, notice that when the ratio between computation and communication is sufficiently high (see Equation 1), communication overhead is completely mitigated in FTPipe by overlapping communication and computation. Thus, to enable dynamic placement of blocks in arbitrary GPUs, it is sufficient to ensure that the blocks have this property, called the *Communication-Computation Overlap* (CCO). The CCO property of a block of operations is easy to verify through profiling, taking into account both the forward edges (activations) and backward edges (gradients). We call a block with the CCO property a CCO block.

The Mixed-pipe partitioning algorithm receives a computational graph of a neural network, which specifies the basic

computational blocks annotated with their memory, communication, and computing requirements.

Then, we proceed as follows:

Step 1: Coarsening: create L pipeline stages. The graph is coarsened by contracting edges and merging their nodes so that its size (typically thousands of nodes) is reduced to L non-input stages whose space requirement is not larger than accelerator memory. Additionally, we ensure that contractions do not create cycles by maintaining a dynamic topological sort [36].

Coarsening begins by eliminating nodes with too high computational load or too high communication load, since both such nodes are bad candidates to become CCO blocks. First, we remove all constants and inputs, later adding and duplicating them if needed to spare redundant communications (this commonly reduces the graph size by half). Second, nodes with weight 0 (typically CPU operations) are eliminated by merging them with their smallest neighbor (computation load-wise). Third, we merge nodes for which the communication volume is above the 95th percentile with their neighbors.

After this processing, we proceed with the creation of L CCO blocks. We tried several approaches. The algorithms involved are very fast, hence several combinations are attempted. We next describe two of the attempts which gave the best results for giant networks according to our experiments.

Systematic creation of CCO blocks. Given a graph (typically of thousands of nodes), we attempt to create the maximal number of CCO blocks by iterating over nodes according to the size of their outgoing edge-cut in ascending order, checking for the CCO property. For a non-CCO node, a backward BFS is initiated, attempting to merge it with its neighbors until it becomes CCO, after which it is not modified anymore, or until no more merges are possible. Next, the algorithm does the same in the opposite direction. It uses forward BFS to iterate over all the unprotected nodes according to their incoming edge-cut size in ascending order, attempting to merge them with their neighbors until they become CCO. In our experiments, we observed that this procedure terminates with a large set of CCO blocks which may now be assigned (obeying space constraints) to arbitrary pipeline stages with no communication penalty, leaving no non-CCO blocks.

Coarsening by type. During tracing, the type and scope of each node are recorded according to its definition in the DNN code. By doing so, large groups of similar nodes can be naturally merged, following user high-level abstractions (e.g. merging all nodes inside 'SelfAttention' block of a Transformer [48]). This usually creates many equal-sized blocks of similar nodes, since giant networks typically contain repeated modules. Our experience shows that this process outputs the best partitioning of CCO blocks (and, consequently, pipeline stages) for giant networks.

From blocks to stages. Once the CCO blocks are created they are packed in the L pipeline stages. In an attempt to localize computation and reduce communication, we finalize the process with a second phase of coarsening. Adjacent CCO blocks are further coarsened together (notice that this maintains their CCO property) while obeying the memory constraint, until the block is sufficiently large to become a pipeline stage. One way to do this is to assign L "center blocks" and repeatedly coarsen them (obeying the memory constraint) in round-robin until no "non-center block" is left. This procedure may repeat several times with different choices of "centers" until a good packing of stages is obtained.

Choosing L. L is chosen at a sweetspot of the tradeoff between large L, which enables better load balancing in Step 2, and small L, which is better staleness-wise. To this end, we try several values of L (e.g. $L = 2P, 3P$) where P is the number of GPUs, and take the option yielding the best pipeline throughput (measured at step 3 below), where L does not exceed a pre-defined upperbound which can be found through experimentation.

Step 2: Load balancing. Here we allocate the L stages to P GPUs in a way that optimizes load-balancing while ensuring that tasks allocated to the same GPU fit in its memory.

The assignment to GPUs can be seen as a classical multiprocessor scheduling problem [23]. While the original problem does not target the pipeline execution setup, in practice, and when communications do not add overhead (the CCO property), the optimal multiprocessor schedule is the one with a balanced distribution of tasks among the processors, meeting the goal of the pipeline assignment. Therefore, we apply the broadly used Longest-Processing Time-First heuristics: the L CCO stages are sorted in a descending order of their computation load, and are assigned to the next least busy GPU with enough memory to run them.

Pipeline stages inside each GPU are created according to the topological order of the stages in the computational graph. A simple algorithm ensures that the depth of the stages is minimized: merge connected components inside each GPU as long as this does not create a cycle with other components. During execution, a stage is invoked by the GPU when its inputs are made available by completion of execution of all previous stages.

Step 3: Refinement We perform fine-grain tuning of the load balance achieved in Step 2. First, the L stages are uncoarsened into their basic blocks. Then, within each pipeline stage, we greedily find blocks which, if moved to an adjacent stage in the pipeline, can improve the throughput, or lower communication or improve load balance (given also the memory constraints).

4.2 Fine-tuning with staleness

Asynchronous pipelines allow faster execution. However, asynchrony introduces staleness. The key problem with staleness is that it can potentially harm final model quality. Note, however, that this is analogous to large batch training [43] which proved to be a popular and useful technique when appli-

cable. In such cases, empirical evidence of acceleration while maintaining model quality is of practical importance even when shown only for some specific (yet important) cases.

Some staleness mitigation techniques introduce high overheads. For example, weight stashing, applied in PipeDream, causes multiple versions of model weights to be stored for the entire course of the pipeline round-trip. This implies that in a pipeline with K workers, the first worker stores up to K versions of weights, which might effectively nullify the memory benefits of model partitioning across the GPUs.

In the case of pipeline execution of fine-tuning tasks, however, we observe that staleness *does not* have a major deteriorating effect on the training quality. There could be several reasons for this phenomenon: (a) Staleness is higher in the initial phase of the training [16], while in fine-tuning we start with pre-trained weights; (b) Fine-tuning usually uses lower learning rates, hence smaller staleness gaps [3]. (c) Momentum [47] exacerbates staleness, but many fine-tuning tasks do not use momentum. Furthermore, the part of staleness caused by momentum can be mitigated using gradient accumulation and momentum weight prediction [12], both can be employed by FTPipe with no memory overhead and a small computational overhead ($< 5\%$, measured on Bert and GPT2). (d) Large models appear to have smooth and slowly varying loss functions [26], meaning that for large models, the stale loss can be a close-enough approximation to the actual loss. Indeed, as can be observed in the experiments Section 5, FTPipe achieves state-of-the-art results with an asynchronous pipeline of up to 16 stages, while avoiding the memory overheads of staleness mitigation implemented in prior works [33].

4.2.1 Checkpointing and recomputation

Checkpointing [6, 11] is a method to reduce memory by keeping only a small state (essentially, partition-border activations and the seed of the random number generator) rather than the whole computation graph. Checkpointing takes place during the forward pass while computing the loss. In this work, we experimented with checkpointing at pipeline stage borders similar to GPipe (In principle, checkpointing can be used at higher granularity). During the backward pass, recomputation reconstructs the parts of the computation graph required for backpropagation.

Checkpointing is used by FTPipe for saving memory during training, but it also mitigates staleness in two ways: First, during backpropagation, recomputation implies that the gradients are computed on up-to-date parameters (but using a stale loss that was computed in a forward pass on an older set of parameters, Table 1). Second, more computations are shifted toward the end of the pipeline where staleness is lower as a result of the last pipeline stage not recomputing, which makes it possible to increase its computational load by around %33.

Setting	Loss	Forward	Backward
Synchronous	θ_t	θ_t	θ_t
Asynchronous	θ_{t-s}	θ_{t-s}	θ_t
PipeDream	θ_{t-s}	θ_{t-s}	θ_{t-s}
FTPipe	θ_{t-s}	θ_t	θ_t

Table 1: Parameter versions during different phases of backpropagation, under different pipeline execution scenarios. θ_j denotes parameters after j optimization steps. A stale parameter with delay of s is denoted θ_{t-s}. Notice that the pass used for loss calculation ('loss') and the pass used to compute activations for backpropagation ('forward pass') induce two separate calculations.

4.3 Profiling

In FTPipe's asynchronous work scheduler (Figure 5) there are two types of stages in the pipeline: (A) the last stage, which does not recompute (B) other stages, which do recompute. For this reason, the execution of the last stage is faster by approximately 33% [15]. Hence, for precisely estimating the load, FTPipe profiles blocks both with and without recomputation. When a block is assigned to the last stage, it uses its profile without recomputations, and vice versa. However, the partitioning itself needs to know which profile to use when the block goes through the coarsening process, and this profile may eventually end up being the wrong one. Thus, several iterations of the partitioning algorithm may be needed to ensure that the correct profile is selected.

5 Evaluation

In this section, we evaluate the performance of FTPipe.
Summary. First, we show that FTPipe significantly improves the existing methods for fine-tuning of giant models.

Second, we show the benefit of Mixed-pipe for both synchronous and asynchronous pipelines. In both cases, despite communicating more and assigning non-consecutive layers of the trained model to workers, we show that Mixed-pipe accelerates execution compared to a Seq-pipe baseline.

Third, we show the benefits of fine-tuning giant models asynchronously, despite the fact that asynchrony introduces staleness. We do so by comparing FTPipe asynchronous pipeline to GPipe synchronous pipeline when both are employed to train the same model using the same partitioning. Our results show that FTPipe asynchronous pipeline is faster to achieve the same top-accuracy of GPipe, across different architectures, datasets, optimizers, and model sizes. Furthermore, our results empirically prove that the additional staleness generated by the increased number of pipeline stages in Mixed-pipe does not degrade its final accuracy.

Implementation. FTPipe has two main components: (a) an automatic neural network partitioning and assignment which builds a data-flow, and (b) a pipelined data-flow execution runtime that executes the partitioned model on GPUs and automatically handles work scheduling and inter-GPU communications. We use CUDA-Aware OpenMPI for inter-GPU communications. Our implementation accepts as the input a neural network implemented in PyTorch [35] (Python API) and the representative training inputs for this network. A partitioned model is automatically generated with everything necessary for our runtime to run it.

5.1 Experimental setup

Hardware. We use a server with 8 RTX2080-Ti GPUs with 11GB, connected via PCIe-III, 64-bit Ubuntu 18.04 with CUDA toolkit 10.2 and cuDNN v7.6.5.

Models and training methodology. We used three different model architectures: Bert [10], GPT2 [39] and T5 [40]. These models represent the typical kinds of NLP architectures used for fine-tuning: Bert is encoder-only, GPT2 is decoder-only, and T5 is encoder-decoder. We took the PyTorch implementations of the models and pretrained weights from HuggingFace [52].

For each combination of (dataset, model, pipeline, partitioning), we chose the number of micro-batches that achieved the best throughput while keeping the mini-batch size constant. For example, GPipe prefers a higher number of micro-batches for a given mini-batch size to increase its pipeline parallelization level. In contrast, FTPipe prefers a lower number of micro-batches as it uses inter-mini-batch parallelism.

Exact hyper parameters are reported in Appendix A.

Tasks and datasets. We use five different learning tasks and six datasets: Natural Language Inference (NLI) using RTE, Word Sense Disambiguation using WiC, Question answering using SQuAD and MultiRC, Boolean Question Answering using BoolQ and Language Modeling using WikiText2 [30, 42, 50, 51].

Our choices are dictated by the following criteria. We choose the task for a given model if that model is known to achieve state-of-the-art results, implying that smaller models are inferior. For example, RTE, WiC, BoolQ, MultiRC were chosen since T5 improved their accuracy considerably. GPT2 for WikiText2 achieves state-of-the-art results. Our fine-tuning improved the advertised results by 6.32 perplexity points on the test set.

We fine-tuned all models and datasets, achieving accuracy comparable to the top published results.

Baselines. Our choice of the baseline is restricted because of the memory requirements of giant models. In particular, neither Single-GPU, data-parallellism [27] nor PipeDream [33] without checkpointing meet these requirements. Hence, we apply checkpointing to PipeDream. Mesh-TensorFlow model-parallel framework [44] failed to run T5, running out of memory, even with checkpointing, FP16, and micro-batch size of 1. This framework was originally used to run T5 on a TPU cluster, but the commodity hardware restrictions impede its use. In general, however, PipeDream already demonstrated the throughput benefits of pipelined execution over the model-parallel one, which applies to our work too.

GPipe is the only framework that successfully runs all the evaluated models. We used synchronous execution with GPipe to set the target accuracy in our experiments.

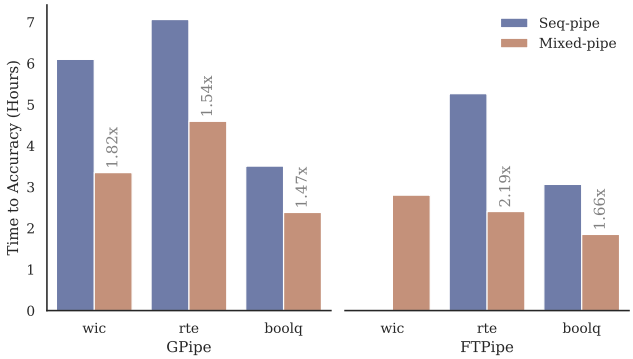

Figure 6: Mixed-pipe speedup over Seq-pipe for reaching top accuracy with T5-3B on three datasets and two pipelines: GPipe (synchronous) and FTPipe (asynchronous).

5.2 End-to-end evaluation

Table 2 shows that *FTPipe outperforms GPipe for all the cases in terms of time-to-top-accuracy*. As expected, the main speedup is due to faster epoch time, but it is not the only factor.

To explain this phenomenon, Figure 7 shows an example of FTPipe training accuracy over time for T5 on RTE. We observe that both systems achieve 75% or higher accuracy at the same time, despite the extra epochs needed for FTPipe. Nevertheless, FTPipe is much faster in the remaining top 25%. The reason is FTPipe staleness, which impacts the training process in the beginning, yet diminishes and becomes negligible as the learning steps shorten [16].

In the following sections, we analyze the two factors contributing to acceleration, namely Mixed-pipe and asynchronous execution.

5.3 Effect of mixed-pipe partitioning

Figure 6 compares the contribution of our partitioning scheme by evaluating different schemes separately for synchronous and asynchronous pipelines for the T5 model on three different tasks and four datasets. Mixed-pipe runs 16 stages (two per GPU). Seq-pipe runs eight stages. We observe that Mixed-pipe is superior to Seq-pipe across all the experiments.

Table 2: Summary of results comparing FTPipe with original GPipe Seq-pipe (synchronous pipeline, one stage per GPU) over 8 GPUs. TTA stands for Time to Accuracy i.e time to reach the top accuracy. We fix top accuracy according to GPipe best results.We denote Mixed-pipe as 'mixed' partitioning. WSD stands for Word Sense Disambiguation, NLI is Natural Language Inference, QA is Question Answering and LM is Language Modeling.

Model	Size	Task	Dataset	Accuracy	Partitioning	Pipeline	Speedup over GPipe Seq-pipe	
							Epoch time	TTA
T5	3B	WSD	WiC	74.92%	mixed	Async	3.28×	2.18×
						Sync	1.82×	1.82×
		NLI	RTE	90.97%	mixed	Async	2.96×	2.94×
						Sync	1.54×	1.54×
		QA	BoolQ	89.05%	mixed	Async	2.85×	1.9×
						Sync	1.47×	1.47×
		QA	MultiRC	85.6 F1, 59.3 EM	mixed	Async	3.05×	3.05×
						Sync	1.3×	1.3×
GPT2	1.5B	LM	WikiText2	12.02 perplexity	sequential	Async	1.6×	1.6×
Bert	340M	QA	Squad	93.3 F1, 87.2 EM	sequential	Async	2.04×	2.04×

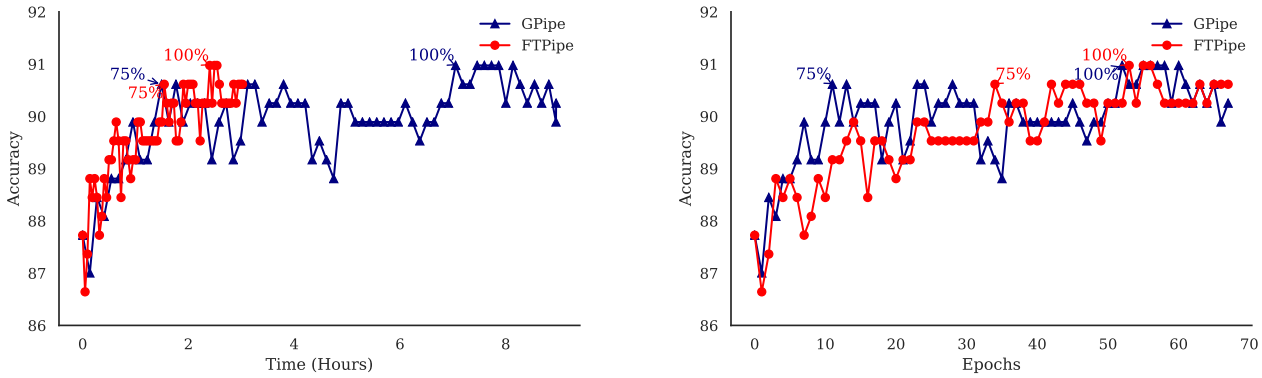

Figure 7: FTPipe acceleration over GPipe for fine-tuning T5-3B with Glue RTE dataset. FTPipe uses Mixed-pipe with 16 stages. GPipe is a synchronous Seq-pipe with 8 stages. We denote the earliest times when at least 75% and 100% of the top accuracy is achieved. Notice that FTPipe is much faster in achieving the last top 25%.

We observed that Mixed-pipe balances GPU memory occupancy across GPUs for both synchronous and asynchronous pipelines, keeping more free memory available, thus allowing a bigger micro-batch size. Most importantly, Mixed-pipe improves both synchronous and asynchronous. We note that while the inter-GPU communication volume in Mixed-pipe is higher than in Seq-pipe, 2.6× for MultiRC and BoolQ, 2.5× for RTE and 3.31× for WiC, the benefits of improved load balance outweigh the associated overheads.

The benefits and trade-offs Mixed-pipe introduces differs for synchronous and asynchronous pipelines:

Synchronous pipeline. For the synchronous setting, using Mixed-pipe can only improve performance and does not affect accuracy. Figure 6 shows that Mixed-pipe improves the time-to-accuracy of GPipe up to 1.8x.

Asynchronous pipeline. For FTPipe asynchronous pipeline, Mixed-pipe may harm accuracy since it adds more pipeline stages, which add staleness. However, in practice, accuracy is not affected. Figure 6 shows that Mixed-pipe both accelerates execution and achieves the same final accuracy result of Seq-pipe. Figure 7 provides the explanation for this phenomena: staleness affects mainly the beginning of the computation, up to the point of reaching 75% of the target accuracy. In this first part of the computation, staleness causes Mixed-pipe to execute more epochs than Seq-pipe, yet they both run at the same wall-clock speed because Mixed-pipe is much faster per epoch. In contrast, during the remaining part of the computations from 75% to 100% of the target accuracy, Mixed-pipe enjoys significant speedup over Seq-pipe with negligible (if at all) effect of staleness as the learning steps gradually shorten [16].

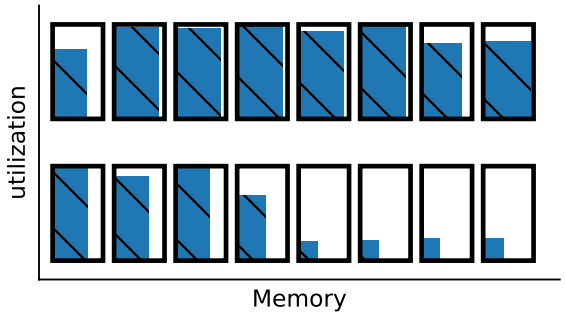

Figure 8: Mixed-pipe load balancing visualization for asynchronous pipelines with T5-3B model on the WiC dataset. Each box is a GPU whose width represents memory consumption and fill represents utilization. Top: Mixed-pipe. Bottom: Seq-pipe.

5.3.1 Load balancing analysis

In Figure 8, we present the load balancing analysis of Mixed-pipe, using T5 as a representative example.

As illustrated in Figure 4, the T5 encoder-decoder architecture with unbalanced inputs sequences for encoder and decoder poses a problem for Seq-pipe. In our experiments, Seq-pipe with memory-unaware partitioning methods resulted in out of memory run-time errors, whereas Seq-pipe with memory-aware partitioning methods resulted in computational load imbalance. In contrast, Mixed-pipe achieves a much better balance both for memory occupancy and computational load.

In summary, *Mixed-pipe improves the performance over Seq-pipe due to two main factors: (1) better computational load balance and (2) better memory balance, thereby enabling larger micro-batch size.*

5.4 Effective fine-tuning with staleness

Table 3 summarises our results for three different model sizes and five different tasks. Each task used the same model partitioning to train GPipe (synchronous) and FTPipe (asynchronous). Note that these results differ from those in Table 2, because here we use the Mixed-pipe partitioning scheme for both FTPipe and GPipe.

In all experiments, FTPipe achieved the same or higher accuracy than GPipe. We note that MultiRC and Squad achieved slightly better results with FTPipe (85.99 F1 60.44 EM; and 93.47 F1 87.38 EM receptively).

5.5 FTPipe vs PipeDream

We implemented PipeDream partitioning based on their open-source GitHub code. To focus on conceptual evaluation, the partitioning algorithm is implemented on top of the technically superior tracing, profiling, compilation, and runtime system of FTPipe, which can handle giant models. Furthermore, it is necessary since only FTPipe supports sharing weights.

Table 3: FTPipe vs GPipe time-to-accuracy, using the *same* Mixed-pipe partitioning. Top accuracy is set by GPipe best results.

Dataset	Model	Parti -tioning	Top Accuracy	Speedup Epoch	Speedup TTA
wic	T5-3B	mixed	74.92%	1.98×	1.19×
rte	T5-3B	mixed	90.97%	1.91×	1.91×
boolq	T5-3B	mixed	89.05%	1.93×	1.29×
multirc	T5-3B	mixed	85.6 F1 59.3 EM	2.34×	2.34×
Wiki -Text2	GPT2-1.5B	seq	12.02 perplexity	1.61×	1.61×
Squad	Bert-340M	seq	93.3 F1 87.2 EM	2.04×	2.04×

T5-3B: PipeDream partitioning of T5-3B for 8 GPUs required around 20 minutes and yielded an infeasible solution with hybrid data- and pipeline-parallel stages, some of which do not fit into GPU memory. In particular, lacking the way to specify memory constraints, PipeDream ended up with more than 1.48B parameters in the last stage (a single 32GB V100 GPU can handle only about 1.3B parameters [41]). It also produced an infeasible solution (OOM) when forced to create a simple pipeline (no data-parallel stages). Note that a pipeline with weight stashing and without gradient accumulation would not be able to train T5-3B on 11*GB* GPUs, even if more GPUs where used, as explained in §4.2.

GPT2-1.5B PipeDream failed to run GPT2-1.5B with 8 GPUs even with checkpointing.

BERT-large: When profiling with micro-batch of size 24, and sequence length 384 (the suggested hyperparameters for SQuAD [10]), PipeDream outputs a purely data-parallel solution (no pipeline) for 2, 4, and 8 GPUs. Such a solution does not fit RTX2080-Ti GPUs even with batch size 1 . Only at micro-batch size 1 PipeDream succeeded in creating a working pipeline for 2 GPUs. For fine-tuning SQuAD with 2 GPUs, FTPipe was 12.7% faster than PipeDream in achieving the same accuracy over 5 seeds (std was 1%), attributed to the enhanced profiling of FTPipe.

For 4 GPUs, FTPipe was 4% faster than PipeDream pipeline[2]. For 8 GPUs, both PipeDream (pipeline) and FTPipe achieve comparable accuracy at comparable times, since the advantage of FTPipe profiling diminishes as the number of pipeline workers increases.

[2]We noticed that PipeDream models data-parallel communication as *completely* concurrent to computation, but in practice, 30% of the communication does not overlap [27]. Changing the modeling led PipeDream to output a pipeline solution.

6 Related Work

Parallel, memory efficient methods for training giant neural networks fall into three categories: Sharded Data Parallelism [41], Intra Layer Model Parallelism [44, 46], and synchronous pipeline parallelism with checkpointing [15]. The communication volumes of intra-layer model-parallelism and shared data-parallelism cannot be overlapped entirely and are too high for commodity interconnects. GPipe [15], similar to sharded data-parallelism, can be made efficient and hide synchronization overheads when using large mini-batches with many micro-batches. However, this is not generally applicable to the batch size used with fine-tuning. Compressing the communication of data-parallelism [28] can significantly reduce its communication volume, making it more suitable for commodity interconnects. For giant networks, however, a combination with another memory reduction technique is required.

In another line of works [16, 17, 19, 54], the backward pass is decoupled and is performed in parallel, but the forward pass is sequential. In Ouroboros [54], shared embedding (aka Tied Weights [18, 38]) of small Transformers were manually placed on the same GPU. In FTPipe, shared weights can also be placed on different GPUs, and it is done with automatic partitioning and with pipelining where communication is overlapped.

PipeDream [33] solves pipeline imbalance problems by incorporating data-parallelism with seq-pipes, allowing a different number of data-parallel GPUs per pipeline stage. This solution requires more GPUs than Mixed-pipe to balance some architectures (see §3.3).

Our experiments with giant models show that the exhaustive search of PipeDream can take a long time for large computational graphs (where FTPipe search took few seconds) and is infeasible for Mixed-pipe. METIS [21] rapidly partitions large computational graphs by multilevel graph partitioning. However, it is focused on optimizing edge-cut under load balance constraints and not on the pipeline throughput. When applied to pipelines, it sometimes unnecessarily creates additional small stages, which Mixed-pipe bounds from above by L.

Several asynchronous model parallelization methods have been recently proposed [34, 53]. These works use Seq-pipes and would suffer from imbalance problems solved by Mixed-pipe.

7 Conclusion

Fine-tuning has the potential to bring the power of huge neural networks to users who do not possess high performance compute resources. In this paper we survey the challenges introduced by this paradigm and take a big step towards solving them by enabling fine tuning on affordable commodity hardware. Our future work will show that FTPipe can scale beyond single-machine boundaries, can achieve higher efficiency by incorporating data-parallel components, and can bring value to non-NLP communities. We hope that the ideas underlying Mixed-pipe can apply beyond fine-tuning, to training in general.

Table 4: Hyper-parameters we used. Accum stands for gradient accumulation steps (micro batches). Batch is mini-batch size. Max steps are the max steps the model was trained for.

Dataset	Pipeline	Batch	Accum	Max steps
squad	ftpipe-seq	24	1	2 epochs
	gpipe-seq	24	8	2 epochs
wikitext2	ftpipe/gpipe-seq	8	8	1 epoch
rte	ftpipe-seq	40	10	4200
	ftpipe-mixed	40	5	4200
	gpipe-seq/mixed	40	10	4200
wic	ftpipe-seq	128	4	17000
	ftpipe-mixed	128	2	17000
	gpipe-seq/mixed	128	8	17000
boolq	ftpipe-seq	20	10	3200
	fpipe-mixed	20	5	3200
	gpipe-seq/mixed	20	10	3200
multirc	ftpipe-seq	8	4	17000
	fpipe-mixed	8	2	17000
	gpipe-seq/mixed	8	8	17000

A Hyper-parameters

Table 4 lists the experiments' hyper-parameters. A checkpoint was saved after every epoch and for WiC every 500 steps. In the T5 and GPT2 experiments, we utilized gradient accumulation to achieve the desired batch size.

T5 experiments used Adafactor optimizer [45] with learning rate of 0.01 and a warm-up of approximately 6% of total fine-tuning steps.

GPT2 experiments used AdamW [29] optimizer with weight decay of 0.01 and a learning rate of 5e-5, decreasing linearly to zero. First and second moment coefficients are 0.9 and 0.999, respectively. We used a mini-batch size of 8 and max sequence length of 1024. We fine-tuned GPT2 for one epoch since further fine-tuning caused over-fitting. Due to large memory consumption caused by the large sequence-length (1024), we used a batch size of 1.

BERT experiments used Adam [22] optimizer with first and second moment coefficients of 0.9 and 0.999 receptively, a learning rate of 3e-5 decreasing linearly to zero. We trained for 2 epochs with mini-batch size of 24 and max sequence length of 384. We loaded weights pre-trained with whole-word-masking.

References

[1] Nvidia ngc. https://NGC.nvidia.com.

[2] Ankur Bapna and Orhan Firat. Simple, scalable adaptation for neural machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1538–1548, Hong Kong, China, November 2019. Association for Computational Linguistics.

[3] Saar Barkai, Ido Hakimi, and Assaf Schuster. Gap-aware mitigation of gradient staleness. In *International Conference on Learning Representations*, 2020.

[4] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[5] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[8] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric Xing. Toward understanding the impact of staleness in distributed machine learning. In *International Conference on Learning Representations*, 2019.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[11] Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[12] Ido Hakimi, Saar Barkai, Moshe Gabel, and Assaf Schuster. Taming momentum in a distributed asynchronous environment. *arXiv preprint arXiv:1907.11612*, 2019.

[13] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[14] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[16] Zhouyuan Huo, Bin Gu, and Heng Huang. Training neural networks using features replay. In *Advances in Neural Information Processing Systems*, pages 6659–6668, 2018.

[17] Zhouyuan Huo, Bin Gu, qian Yang, and Heng Huang. Decoupled parallel backpropagation with convergence guarantee. volume 80 of *Proceedings of Machine Learning Research*, pages 2098–2106, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[18] Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462*, 2016.

[19] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.

[20] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[21] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[22] Diederik P Kingma and Lei Ba. J. ADAM: a method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[23] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[24] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[25] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[26] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31:6389–6399, 2018.

[27] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[28] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[29] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[30] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[31] Orlando Moreira, Merten Popp, and Christian Schulz. Graph partitioning with acyclicity constraints. *arXiv preprint arXiv:1704.00705*, 2017.

[32] Orlando Moreira, Merten Popp, and Christian Schulz. Evolutionary multi-level acyclic graph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 332–339, 2018.

[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[34] Jay H Park, Gyeongchan Yun, Chang M Yi, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism. *arXiv preprint arXiv:2005.14038*, 2020.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 2019.

[36] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)*, 11:1–7, 2007.

[37] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 7–14, Florence, Italy, August 2019. Association for Computational Linguistics.

[38] Ofir Press and Lior Wolf. Using the output embedding to improve language models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 157–163, Valencia, Spain, April 2017. Association for Computational Linguistics.

[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[41] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

[42] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.

[43] Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. M. *arXiv preprint arXiv:1811.03600*, 2018.

[44] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[45] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. volume 80 of *Proceedings of Machine Learning Research*, pages 4596–4604, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

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

[47] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[49] Mohamed Wahib, Haoyu Zhang, Truong Thao Nguyen, Aleksandr Drozd, Jens Domke, Lingqi Zhang, Ryousei Takano, and Satoshi Matsuoka. Scaling distributed deep learning workloads beyond the memory capacity with karma. *arXiv preprint arXiv:2008.11421*, 2020.

[50] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, pages 3266–3280, 2019.

[51] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[52] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[53] An Xu, Zhouyuan Huo, and Heng Huang. On the acceleration of deep learning model parallelism with staleness. In *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[54] Qian Yang, Zhouyuan Huo, Wenlin Wang, and Lawrence Carin. Ouroboros: On accelerating training of transformer-based language models. In *Advances in Neural Information Processing Systems*, pages 5520–5530, 2019.

[55] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.

[56] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-SGD for distributed deep learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2016.

[57] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. In *International Conference on Machine Learning*, pages 4120–4129. PMLR, 2017.