mLoRA: Fine-Tuning LoRA Adapters via Highly-Efficient Pipeline Parallelism in Multiple GPUs

Zhengmao Ye*
yezhengmaolove@gmail.com
Sichuan University

Dengchun Li*
mikecovlee@163.com
Sichuan University

Zetao Hu*
vinkle-hzt@outlook.com
Sichuan University

Tingfeng Lan
tafflan2001@gmail.com
Sichuan University

Jian Sha
jian.sha@antgroup.com
Ant Group

Shicong Zhang
zsc@ix.cn
Zhejiang New Internet Exchange Center

Lei Duan
leiduan@scu.edu.cn
Sichuan University

Jie Zuo†
zuojie@scu.edu.cn
Sichuan University

Hui Lu
hui.lu@uta.edu
The University of Texas at Arlington

Yuanchun Zhou
zyc@cnic.cn
CNIC, Chinese Academy of Science

Mingjie Tang†
tangrock@gmail.com
Sichuan University

ABSTRACT

Transformer-based large language models (LLMs) have demonstrated outstanding performance across diverse domains, particularly in the emerging *pretrain-then-finetune* paradigm. LoRA, a parameter-efficient fine-tuning method, is commonly used to adapt a base LLM to multiple downstream tasks. Further, LLM platforms enable developers to fine-tune multiple models and develop various domain-specific applications simultaneously. However, existing model parallelism schemes suffer from high communication overhead and inefficient GPU utilization.

In this paper, we present mLoRA, a parallelism-efficient fine-tuning system designed for training multiple LoRA across GPUs and machines. mLoRA introduces a novel LoRA-aware pipeline parallelism scheme that efficiently pipelines LoRA adapters and their distinct fine-tuning stages across GPUs and machines, along with a new LoRA-efficient operator to enhance GPU utilization. Our extensive evaluation shows that mLoRA can significantly reduce average fine-tuning task completion time, e.g., by 30%, compared to state-of-the-art methods like FSDP. More importantly, mLoRA enables simultaneous fine-tuning of larger models, e.g., two Llama-2-13B models on four NVIDIA RTX A6000 48GB GPUs, which is not feasible for FSDP due to high memory requirements. Hence, mLoRA not only increases fine-tuning efficiency but also makes it more accessible on cost-effective GPUs.

*These authors contributed equally to the paper
†Corresponding author

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/TUDB-Labs/mLoRA.

1 INTRODUCTION

Transformer-based, pre-trained large language models (LLMs), such as Gemma [64], LLaMA [65], and Phi-3[8] have expanded their reach beyond natural language processing to a broad range of domain-specific tasks. This is achieved by adapting pre-trained LLMs for downstream tasks via *fine-tuning*, which enhances model performance for a particular task with brief training on task-specific data [15, 49]. Examples of this adaptation include translating natural language questions into SQL queries for relational databases [27], converting heterogeneous data lakes into structured, queryable tables [10], and others [26, 34, 37, 47, 54].

As the size of LLMs grows exponentially – rising from hundreds of billions to the anticipated trillions of model parameters [68] – fine-tuning these models using traditional *full-weight* approaches, which require updating all parameters, becomes very expensive. Instead, Parameter-Efficient Fine-Tuning (PEFT) methods [28], including partial [9, 24, 78], additive [11, 30, 38, 59], and reparameterized [31] fine-tunings, have been developed. They train a much smaller set of parameters, thus cutting training costs while maintaining performance levels comparable to full-weight fine-tuning. Low-Rank Adaptation (LoRA) [31], one of the most effective PEFT methods [77], freezes the parameters of an LLM while updating pairs of low-rank matrices with far fewer parameters, namely *adapter weights*. Models fine-tuned with LoRA not only match but also

exceed the performance of fully fine-tuned models while remaining extremely lightweight, e.g., requiring less than 1% of trainable parameters [28]. The cost-effectiveness and high performance of LoRA have spurred the development of numerous custom LLMs. For example, as of Oct. 2024, approximately 143,920 LoRA adapters have been published on HuggingFace [4], with most of them exhibiting notable performance in its specific domain [13, 40, 71].

While recent attention has largely focused on LLM serving, such as resource efficiency, serving latency, scalability, scheduling, fairness, and multi-tenancy [17, 29, 36, 62, 63, 69, 70, 73], less attention has been paid to addressing an equally important question: *how to effectively and efficiently build these fine-tuned variants?* Unlike training an LLM from scratch, which can require thousands of GPUs and days of time [12, 61], lightweight LoRA enables a single GPU to build multiple model variants simultaneously, with even greater capacity when using multiple GPUs. Meanwhile, concurrently fine-tuning multiple adapters has become increasingly crucial: LLM platforms [2, 3, 6] enable developers to fine-tune multiple models and develop various domain-specific applications simultaneously; for individual developers, selecting multiple sets of hyperparameters either manually or automatically [67] by fine-tuning multiple adapters can quickly reveal the best-performing adapter.

However, the unique characteristics of LoRA present key challenges for parallel fine-tuning LoRA adapters. It can be envisioned that *sharing the same read-only base model with identical pre-trained weights* could facilitate the parallel training of multiple adapters. This reduces GPU memory footprint and enhances training parallelism. Nevertheless, when fine-tuning massive LoRA adapters exceeds the capacity of a single GPU, multiple GPUs become necessary; distributing a base model across GPUs involves *model parallelism*, which partitions the base model's parameters and adapters and distributes them among these GPUs. Unfortunately, existing model parallelism methods, such as tensor parallelism [33, 53] and pipeline parallelism [25, 32], are plagued by *high communication overhead* due to the need for inter-GPU synchronization or *inefficient GPU utilization* caused by pipeline bubbles. Moreover, the small size of LoRA adapters exacerbates the issue – training small LoRA adapters in parallel results in frequent GPU kernel launches, which can substantially increase the total training time.

To overcome these challenges, we present **mLoRA**, a fine-tuning system designed and developed for efficiently fine-tuning LoRA adapters across multiple GPUs and machines. The key goal of mLoRA is to achieve high fine-tuning performance by fully utilizing multi-GPU resources, including both computation and memory.

mLoRA first introduces a novel *pipeline parallelism* mechanism called LoRAPP, which ensures low communication overhead, high parallelism, and improved GPU efficiency for multi-LoRA, multi-GPU fine-tuning. LoRAPP capitalizes on the observation that although different LoRA adapters share the same base model, they can be trained independently without computational dependencies. This enables mLoRA to avoid pipeline stalls by freely and concurrently scheduling distinct training stages (e.g., forward and backward propagation) of different fine-tuning tasks, thus eliminating pipeline bubbles (i.e., *zero* bubbles). Further, mLoRA boosts GPU efficiency with a new *operator*, BatchLoRA. This operator consolidates multiple LoRA fine-tuning tasks into a large batch and performs collective matrix multiplication operations for all

(a) LoRA Approach **(b) BatchLoRA Approach**
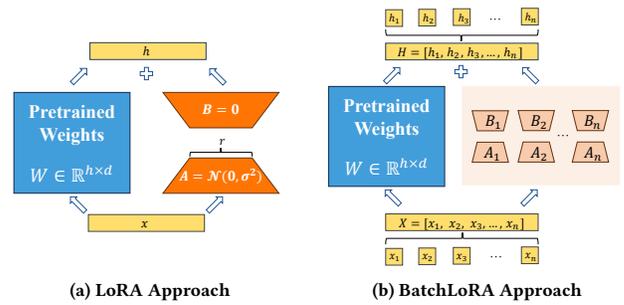
Figure 1: Sharing pre-trained model weights for fine-tuning multiple LoRA adapters with reduced overhead.

involved adapters. This approach enhances GPU utilization and reduces kernel launch overhead while maintaining model quality.

We have evaluated mLoRA by fine-tuning multiple adapters on various publically available LLMs of different sizes, e.g., TinyLlama-1.1B [75], Llama-2-7B, and 13B [66]. Experiments demonstrate that mLoRA significantly reduces the completion time for fine-tuning tasks. For instance, it achieves a reduction in fine-tuning time by up to 45% for the Llama-2-7B model in fp32 precision across four NVIDIA RTX A6000 48GB GPUs, compared to state-of-the-art methods like FSDP [79], which is an industry-grade parallel LLM training strategy. Moreover, mLoRA enables the simultaneous fine-tuning of *larger* models, e.g., two Llama-2-13B models in fp32 precision with 4 NVIDIA RTX A6000 48GB GPUs, while FSDP cannot due to higher memory requirements. With its high fine-tuning efficiency and low cost, mLoRA addresses the critical issue of the scarcity and expense of high-end GPUs and has been deployed in the production environment at AntGroup, where it reduces the time for selecting optimal hyperparameters for LLM models by 30%.

2 BACKGROUND AND MOTIVATION

2.1 LoRA-based LLM Finetuning

Training an LLM from scratch demands extensive computational resources over days of time, often utilizing thousands of GPUs and incurring significant financial costs [12, 61]. In contrast, fine-tuning pre-trained language models (PLMs) has made LLM benefits more accessible. Organizations like Meta and Google provide their PLMs, such as LLaMA [65] and Gemma [64], to the public. Fine-tuning these models for various downstream tasks is effective [56] and offers a more cost-efficient way to harness LLM capabilities.

Conventionally, full-weight fine-tuning of large-scale pre-trained models requires updating all parameters, which often incurs prohibitive computational costs. In contrast, Parameter-Efficient Fine-Tuning (PEFT) methods [48] selectively update only a small subset of parameters, significantly reducing computational and memory resources. LoRA [31], a state-of-the-art PEFT technique, achieves efficient fine-tuning by freezing the pre-trained model and only updating low-rank additive matrices with far fewer parameters, as expressed in Equation 1.

$$h = xW' = x(W + AB) = xW + xAB \qquad (1)$$

Where x denotes the input data, $W \in \mathbb{R}^{h \times d}$ represents the frozen pre-trained model weights, and $A \in \mathbb{R}^{h \times r}$ and $B \in \mathbb{R}^{r \times d}$ are two low-rank decomposition matrices, with rank $r \ll \min(h, d)$. Additionally, techniques such as quantization [22], which compress the parameters to low precision, and activation checkpointing [18], which recompute activation values during gradient computation, can be employed to reduce GPU memory footprint further.

Figure 1(a) shows a typical way to train a single LoRA adapter from a frozen PLM. When training multiple LoRA adapters simultaneously, it makes intuitive sense to *share the same read-only base model* among them to reduce the GPU memory footprint, as shown in Figure 1(b). A naive implementation for such simultaneous fine-tuning is listed in Algorithm 1: It keeps the base model on the GPU throughout the entire training process for all LoRA tasks, *only* swapping the adapter weights for each task sequentially.

Algorithm 1 Simply train multiple LoRAs, PyTorch-like.

```
for adapter, data in fine_tuning_task:
    A, B = adapter # swap in the low-rank matrix A and B
    output = data @ W + data @ A @ B
    loss = loss_fn(data, output)
    loss.backward()
```

Algorithm 2 Use the BatchLoRA to train, PyTorch-like.

```
datas = [data for _, data in fine_tuning_task]
adapters = [adapter for adapter, _ in fine_tuning_task]
output = datas @ W # just call once
output += BatchLoRA.apply(datas, adapters)
loss = loss_fn(data, output)
loss.backward()
```

2.2 Multi-LoRA Finetuning across Multi-GPU

When the need to fine-tune multiple high-precision LoRA adapters or the read-only base model is too large and exceeds the capacity of a single GPU – mainly due to limited GPU memory and/or computation – parallelization through multiple GPUs is necessary. Two common parallelism methods are *data parallelism* (DP) [43] and *model parallelism* [53]. Data parallelism requires each GPU to store a complete set of model parameters, which is inefficient and even impossible for LLM training/fine-tuning when the model size is large and the GPU memory is small.

To address this, model parallelism partitions and distributes model parameters across GPUs. *Tensor parallelism* (TP), one of the representative model parallelism strategies, splits a tensor (e.g., a vector or matrix) in the model into multiple chunks along a specific dimension. Each GPU only holds one chunk of the tensor and computes partial results based on the allocated tensor chunk. All partial results are combined into the final result through collective communication methods, such as all-reduce or all-gather. However, this approach introduces significant synchronization overhead, particularly in inter-machine setups, where limited communication bandwidth can substantially slow down LLM training.

To mitigate this, pipeline parallelism (PP) divides the model into sequential groups, each containing one or more layers of the model. Each GPU handles a separate group and computationally depends on its previous GPU, which manages the preceding group. Consequently, input data is processed in a sequential, pipelined manner, passing through the dependent GPUs. PP reduces communication overhead by transmitting only the results of the last layer in a group
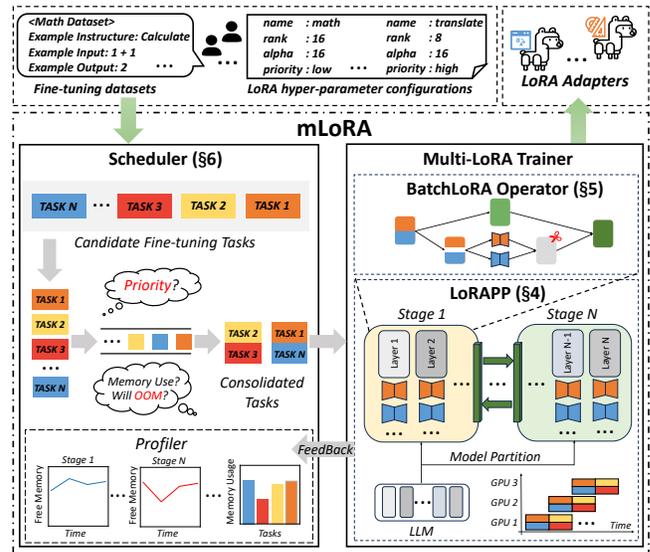

Figure 2: Overview of mLoRA.

between adjacent GPUs rather than synchronizing the intermediate results of each tensor within each layer. Nevertheless, GPU idle times can be significant due to the computational dependencies of PP. Solutions like PipeDream [51] and PipeMare [72] relax dependency constraints, e.g., using mismatched weight versions between forward and backward propagation, to reduce pipeline bubbles. However, recent works [42, 45, 50] suggest that these methods may lead to lower convergence performance. In the context of LoRA-based fine-tuning, we have two key observations:

Observation 1: Unlike existing model parallelism strategies that require pipelining the *dependent* processing stages when training a single LLM, the *independent nature* of fine-tuning *multiple* LoRA adapters, despite sharing the same base model, can enable more efficient processing and greater parallelism. For example, we can populate a fully occupied fine-tuning pipeline across multiple GPUs and machines by scheduling distinct training stages for separate LoRA adapters *concurrently*. Further, by overlapping GPU communication and computation across separate stages, we can effectively hide I/O latencies and maximize overall GPU efficiency.

Observation 2: The overhead from calling the API to launch GPU kernel functions can be substantial. This is particularly true when we fine-tune numerous small LoRA adapters with a naive parallel scheme like Algorithm 1, which leads to frequent kernel launches and high overhead, e.g., accounting for up to 10% of the total training time. A promising solution to mitigate this overhead, as illustrated in Figure 1(b) and Algorithm 2, is to consolidate the training data from multiple fine-tuning tasks into a larger batch. By performing matrix operations for all involved adapters collectively, we can achieve the same results as executing multiple fine-tuning tasks sequentially (as that in Algorithm 1) but with fewer GPU kernel launchers and reduced overall training time.

3 DESIGN OF MLORA

The limitations of existing model parallelism methods and the observations in Section 2.2 motivate us to design mLoRA, a new fine-tuning system for the efficient training of multiple LoRA adapters. In this section, we first present an overview of mLoRA, including its key design objectives and fine-tuning workflow, and then detail the key techniques that underpin mLoRA.

3.1 Overview

Design objectives: mLoRA is developed to fine-tune multiple LoRA adapters efficiently across one or multiple (cost-effective) GPUs. It optimizes training throughput and resource utilization via two new techniques: 1) LoRA-aware pipeline parallelism, LoRAPP (§3.2), and 2) LoRA-efficient training operator, BatchLoRA (§3.3).

Architecture Overview: As illustrated in Figure 2, mLoRA consists of two main components: 1) A *multi-LoRA trainer* capable of simultaneously handling multiple LoRA fine-tuning tasks while conducting runtime optimization via BatchLoRA and LoRAPP technologies. 2) A task scheduler that can choose a batch of fine-tuning tasks based on user demands and metrics from the performance profiler, e.g., to schedule tasks to maximize GPU resource utilization and minimize the out-of-memory (OOM) issues.

Specifically, users initiate requests to mLoRA, providing hyper-parameter configurations for the LoRA adapters and the datasets used for fine-tuning. Based on this, mLoRA generates candidate tasks with their initial configurations and places them in a candidate task queue. Then, the task scheduler chooses tasks from the candidate task queue for parallel training (§ 3.2 and § 3.3) based on various scheduling factors (§ 3.4), such as the memory footprint and task priority. During the training, the multi-LoRA trainer provides performance metrics to the profiler, including the actual memory usage of the current task. The profiler then uses this information to keep revising its memory estimation model (§ 3.4), enabling more precise assessments of memory requirements for future tasks.

3.2 Multi-LoRA Training Parallelism

3.2.1 LoRA-aware Pipeline Parallelism (LoRAPP). As discussed in Section 2, pipeline parallelism can lead to idle periods and inefficiencies due to computational dependencies between GPUs. For example, in Figure 4 (a), the traditional pipeline parallel algorithm GPipe [32] requires $GPU0$ to wait for $GPU1$ to complete $B1$ before $GPU0$ can execute $B1$, creating idle times for $GPU0$, known as *pipeline bubbles*. Drawing on Observation 1 (§ 2.2), we propose LoRAPP, a novel pipeline parallelism strategy to optimize fine-tuning multiple LoRA tasks by reducing or eliminating these bubbles.

Base workflow of LoRAPP. As illustrated in Figure 3, the workflow of LoRAPP comprises two main stages: In the *preparation* stage, the model partition module partitions the pre-trained base LLM – comprising consecutive transformer decoder layers – into separate groups and allocates these groups to available GPUs (e.g., one group for each GPU). Note that model partitioning is not the focus of mLoRA and has been extensively covered in recent work [25, 32, 51]; LoRAPP adopts the partitioning approach from GPipe to ensure that each group has an equal computational load. In the *training* stage, following mLoRA's scheduling scheme (§ 3.4),
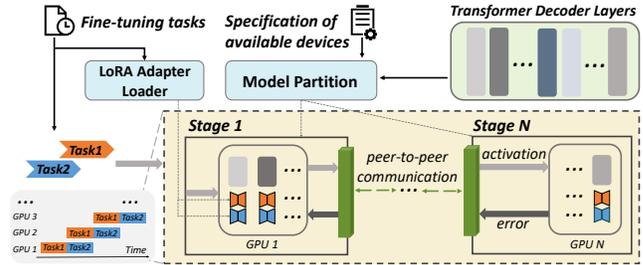

Figure 3: The workflow of LoRAPP.

a set of fine-tuning tasks is selected for parallel training and populating the multi-GPU pipeline: 1) For each LoRA adapter, the LoRA adapter loader module allocates a small amount of memory on each GPU to store a portion of the adapter's weights associated with the linear layers of the base model. These weights are randomly initialized as described in LoRA [31]. 2) After initialization, each GPU performs *forward propagation* using activation values received from its previous GPU's forward propagation. 3) After forward propagation, each GPU performs *backward propagation* using error values received from its next GPU's backward propagation.

During the pipelined processing, the first and last GPUs operate slightly differently from others: 1) The first GPU in the pipeline receives the training data for a fine-tuning task to initiate the training process and does not need to send error values. 2) The last GPU computes the loss using the activation values and then begins the backward propagation, without needing to send activation values. Once the fine-tuning task is finished, the weights of its LoRA adapters are saved (e.g., to persistent storage), and the allocated memory spaces can be released and used for new tasks.

Achieving Zero Bubbles in LoRAPP. A key goal of LoRAPP is to reduce or eliminate pipeline bubbles and achieve high efficiency in pipelined fine-tuning. Existing pipeline approaches, like GPipe [32] as illustrated in Figure 4 (a), address this by dividing a mini-batch into smaller micro-batches to populate the pipeline during each training step or iteration. However, to ensure model convergence, the mini-batch gradient descent algorithm [41] requires that the pipeline waits for gradients from all micro-batches within a mini-batch to accumulate before applying them. This *stop-and-wait* synchronization introduces pipeline bubbles. One approach is to increase the number of micro-batches to alleviate the pipeline bubbles to some extent, but the number of micro-batches is constrained by the mini-batch size. Moreover, larger mini-batch sizes can negatively impact model convergence [14, 20], further restricting the mini-batch size. Another approach is to launch multiple GPipe instances, but coordinating multiple instances to utilize their pipeline bubble is difficult, and each instance needs to load a copy of the same read-only base model, leading to memory waste. As a result, it is hard for existing pipeline parallel approaches to achieve zero pipeline bubbles while ensuring model convergence.

In contrast, LoRAPP reduces the pipeline bubble to *zero* based on Observation 1 (§ 2.2): Since each LoRA adapter independently accumulates and applies gradients, there is no need to synchronize gradients between different LoRA adapters. Thus, LoRAPP can use mini-batches from different LoRA adapters to populate the pipeline.
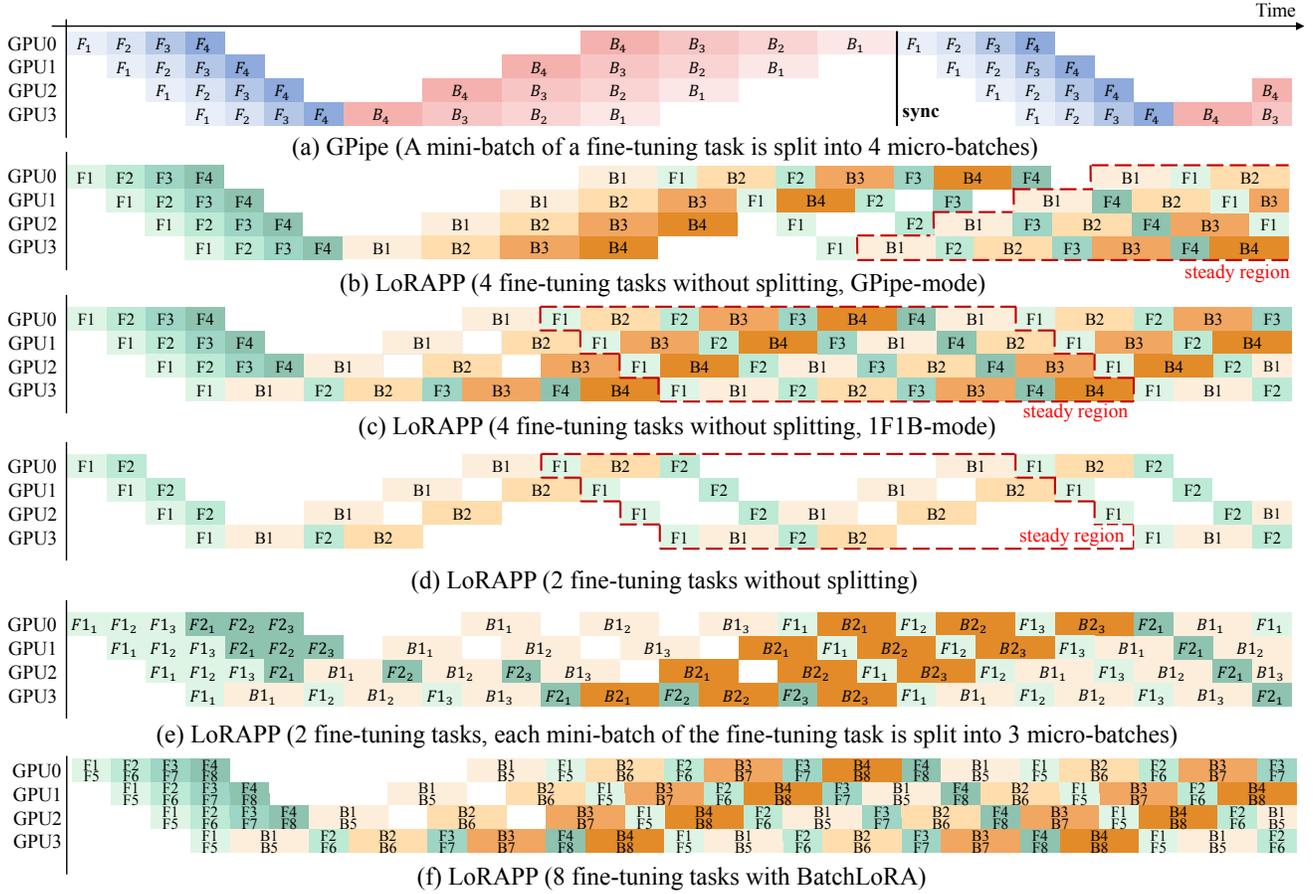
Figure 4: (a) GPipe. The training data of the fine-tuning task are divided into four micro-batches within a mini-batch. Here, F_i represents the forward propagation of the ith micro-batch, while B_i represents its backward propagation. GPipe requires all micro-batches of the same mini-batch to be completed before proceeding to the next mini-batch. (b) (c) (d) LoRAPP without mini-batch splitting. Fi represents the forward propagation of the ith LoRA adapter, while Bi is its backward propagation. (e) LoRAPP with mini-batch splitting. Fi_j represents the forward propagation of the jth mini-batch, into which the macro-batch of training data for the ith LoRA adapter is divided, while Bi_j represents its backward propagation. (f) LoRAPP with BatchLoRA.

For example, in Figure 4 (b), after $GPU0$ completes the forward propagation $F1$ of LoRA adapter 1, it immediately begins the forward propagation $F2$ of LoRA adapter 2. When $GPU0$ completes the backward propagation $B1$ of LoRA adapter 1, it needs to choose between executing $F1$ and $B2$. To minimize the pipeline bubble, we prioritize executing $F1$ to maximize the number of different LoRA adapters being fine-tuned simultaneously. Similarly, when $GPU3$ completes the forward propagation $F1$ of LoRA adapter 1, it needs to choose between executing $F2$ and $B1$. Since backward propagation can free up a significant amount of memory used for activations, optimizations, and weight gradients, we prioritize executing backward propagation to free up memory. By default, we adopt the 1F1B-mode (as shown in Figure 4 (c)), which has the same steady region as GPipe-mode but saves memory.

One problem remains: LoRAPP cannot achieve zero bubbles with fewer fine-tuning tasks, as shown in Figure 4 (d). To overcome this, as illustrated in Figure 4 (e), within the same LoRA adapter, LoRAPP

adopts the same strategy as GPipe, which divides the mini-batch into multiple (e.g., three) micro-batches to reduce the bubbles.

The independence of training multiple LoRA adapters also enables the opportunity to overlap GPU communication and computation. As illustrated in Figure 5, since there is no dependency between the ith and jth LoRA adapters, while the ith LoRA adapter's backward propagation B_i is being executed on GPU $K + 1$, it can simultaneously receive the jth LoRA adapter's forward propagation Fj from GPU k. Such overlapping can greatly hide the I/O latency from GPU computation, further improving the efficiency of LoRAPP. More concretely, we create three independent and concurrent running CUDA streams for each GPU, each dedicated to receiving, sending, and computing data.

3.2.2 Cost Analysis of LoRAPP. To quantify the overhead introduced by pipeline bubbles in LoRAPP, we define the *bubble ratio* as the ratio of GPU *idle time* to the total *runtime* of the pipeline.
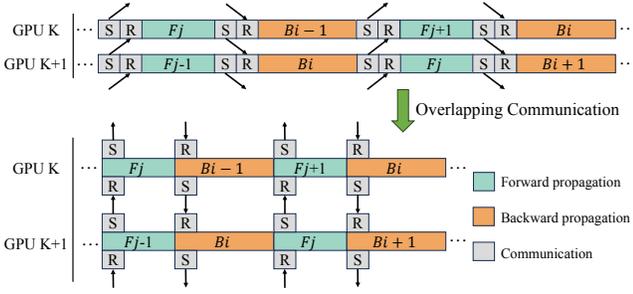
Figure 5: Overlapping communication in LoRAPP. Fi represents the forward propagation of the ith LoRA adapter, while Bi is its backward propagation.

Bubble ratio in LoRAPP. As shown in Figure 4 (c), each subsequent region repeats the steady region, so we can measure the bubble ratio by focusing on one steady region. We define the forward propagation time as T_f, and the backward propagation time as T_b, with a total of D GPUs training L tasks simultaneously. Then, the total time of the steady region is $D^2(T_f + T_b)$, and the idle region is $max\{D(T_f + T_b)(D - L), 0\}$.

Therefore, the bubble ratio of LoRAPP without using mini-batch splitting is $max\{(D - L)/D, 0\}$. Similarly, we can obtain the bubble ratio of GPipe as $(D - 1)/(N + D - 1)$, where N represents the number of micro-batches. This means that if the number of LoRA adapters trained in parallel is greater than or equal to the number of GPUs, LoRAPP can fill the pipeline to fully utilize all GPUs. As mentioned earlier, the number of macro-batches N usually has a small value, preventing GPipe from achieving a zero bubble ratio.

When the system has more GPUs and fewer LoRA adapters for fine-tuning, LoRAPP achieves a relatively high bubble ratio. To further decrease the bubble ratio, as shown in Figure 4 (e), LoRAPP adopts the same strategy as GPipe. This way, its bubble ratio is $max\{(D - 1 + N - L \times N)/(D + N - 1), 0\}$.

Communication cost. As shown in Figure 5, mLoRA can overlap communication and computation. For each GPU, without communication overlap, the latency per training step (i.e., one forward and one backward) is $2C_S + 2C_R + D_F + D_B$, where C_S is the communication latency for sending data, C_R is the communication latency for receiving data, D_F is the computation latency for forward propagation, and D_B is the computation latency for backward propagation. With communication overlap, the latency reduces to $max\{C_S, C_R, D_F\} + max\{C_S, C_R, D_B\}$.

Memory usage. One key difference between LoRAPP and GPipe when simultaneously training L number of LoRA adapters is that LoRAPP shares the same read-only base model among these adapters, whereas GPipe needs to launch multiple instances to load the base model separately. Therefore, LoRAPP saves $(L - 1)W_\theta$ memory, where W_θ is the size of the pre-trained model.

3.3 Multi-LoRA Training Operator

With zero bubbles and hiding communication latency, LoRAPP (§ 3.2) achieves efficient pipelined fine-tuning across multiple GPUs. However, we observe that the pipelined GPUs remain not fully utilized. One reason lies in that, unlike complex tasks (e.g., full fine-tuning), each LoRA fine-tuning task (i.e., forward or backward propagation) performed by a GPU is relatively simple and cannot fully exploit the parallel processing capabilities of the GPU. As shown in Figure 4 (b), though theoretically, *four* fine-tuning tasks can reduce the pipeline's bubble to zero with *four* GPUs (i.e., according to the bubble ratio in § 3.2.2), a single GPU only uses part of its computation resources practically. For example, with the workload and single-machine multi-GPU setup in Section 4.1, using Llama-2-7B as the base model with four fine-tuning tasks, the average GPU utilization is 83%, and the average memory utilization is only 30%.

To further improve GPU efficiency and utilization, one intuitive approach is to maximize the number of distinct fine-tuning tasks in the LoRAPP pipeline by scheduling as many LoRA adapters as possible. Note that the maximum number of LoRA adapters each GPU can handle is constrained by its memory size. However, as highlighted in Observation 2 from Section 2.2, the overhead from calling CUDA APIs to launch GPU kernel functions can be nontrivial when training numerous small LoRA adapters (with Algorithm 1). To address this, mLoRA introduces a new operator, BatchLoRA, which allows multiple LoRA adapters to *concurrently* share the pre-trained base model with reduced kernel launch overhead.

3.3.1 BatchLoRA Operator. As illustrated in Algorithm 2 and Figure 1(b), *BatchLoRA* consolidates the training data for a selected number of LoRA fine-tuning tasks into a single large batch (i.e., a large matrix) during each training iteration. Therefore, multiple LoRA adapters can share the same pre-trained model and participate in training *concurrently* – instead of sequentially like Algorithm 1.

We use Figure 1 (b) as the running example. Suppose a set of fine-tuning tasks, denoted as $T_1, ..., T_n$. Each fine-tuning task, T_i, consists of the fine-tuning input data represented as x_i, along with the low-rank weights A_i and B_i of the LoRA adapters. Note that, all the fine-tuning tasks share the same pre-trained weights W. Formally, given the input data x_i for the i-th fine-tuning task and the output data h_i, the consolidated input data $X = (x_1^\top, \ldots, x_n^\top)^\top$. The calculation formula for forward propagation is shown as Formula 2.

$$H = \begin{pmatrix} h_1 \\ \vdots \\ h_n \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} W + \begin{pmatrix} x_1 A_1 B_1 \\ \vdots \\ x_n A_n B_n \end{pmatrix} = XW + \begin{pmatrix} x_1 A_1 B_1 \\ \vdots \\ x_n A_n B_n \end{pmatrix} \quad (2)$$

For backward propagation, according to Formula 2, we derive the gradient formula for each tensor involved in the computation as Formula 3 and 4. Note that W, i.e., the frozen pre-trained weights, does not require training, so its gradients do not need to be computed.

$$\begin{pmatrix} \nabla A_1 \\ \vdots \\ \nabla A_n \end{pmatrix} = \begin{pmatrix} x_1^\top \nabla h_1 B_1^\top \\ \vdots \\ x_n^\top \nabla h_n B_n^\top \end{pmatrix}, \quad \begin{pmatrix} \nabla B_1 \\ \vdots \\ \nabla B_n \end{pmatrix} = \begin{pmatrix} A_1^\top x_1^\top \nabla h_1 \\ \vdots \\ A_n^\top x_n^\top \nabla h_n \end{pmatrix} \quad (3)$$

$$\nabla X = \begin{pmatrix} \nabla x_1 \\ \vdots \\ \nabla x_n \end{pmatrix} = \nabla H W^\top + \begin{pmatrix} \nabla h_1 B_1^\top A_1^\top \\ \vdots \\ \nabla h_n B_n^\top A_n^\top \end{pmatrix}, \nabla H = \begin{pmatrix} \nabla h_1 \\ \vdots \\ \nabla h_n \end{pmatrix} \quad (4)$$

Therefore, based on Formula 2 and 4, we can find that after the training data is consolidated, we only need to launch the matrix multiplication operation XW and $\nabla HW^{\top}$ once on the GPU, rather than launching the matrix multiplication operation x_iW and $\nabla h_iW^{\top}$ for each LoRA adapter, thereby reducing the overhead of kernel launches. Note that training with consolidated data does not affect the model performance and isolation between different fine-tuning tasks, since each LoRA adapter only uses the specific portion of the training data that belongs to this adapter for computation.

Workflow of BatchLoRA. mLoRA follows existing reverse-mode automatic differentiation and gradient computation, implemented through computational graphs [55]. It automatically determines a backward propagation computational graph based on the forward propagation computational graph (defined by the user) and then computes the gradients through this graph. For example, the computational graph of the BatchLoRA operator, as shown in Figure 6, consists of two parts: the forward propagation defined by the user (i.e., the left diagram) and the backward propagation automatically determined (i.e., the right diagram).

For BatchLoRA's forward propagation, the consolidated input data X is used to compute intermediate results $Y = XW$ with the frozen pre-trained weights W. Then, since the consolidated data X represents the training data for multiple LoRA adapters, we need to split it into multiple chunks $x_1, \ldots, x_n$ to make sure that each chunk represents the training data for its corresponding LoRA adapter, i.e., ensuring isolation among tasks (and their users). These data are separately computed with their respective LoRA adapters, resulting in intermediate results $L_i = x_iA_iB_i$. Finally, the intermediate results L_i are added to Y based on their original positions before splitting to obtain the final output H.

The backward propagation of BatchLoRA consists of two parts. In the first part, the gradients of the LoRA adapter ∇A_i and ∇B_i are computed as follows: Based on the split-position information during the forward propagation, the input backward propagated error ∇H is split into multiple chunks $\nabla h_1, \ldots, \nabla h_n$, where each chunk represents the input backward propagated error for each LoRA adapter. Then, according to Formula 3, the gradients for each LoRA adapter are computed separately. In the second part, the output backward propagated error values ∇X are computed. According to Formula 4, first, the intermediate value $\nabla Y = \nabla HW^{\top}$ is computed; then the backward propagated error for each LoRA adapter is computed as $\nabla x_i = \nabla h_iB_i^{\top}A_i^{\top}$. Finally, the backward propagated errors $\nabla x_1, \ldots, \nabla x_n$ are consolidated into an intermediate value through the derivative of the split operator and added to ∇Y to generate the final output backward propagated error ∇X.

Graph pruning. The backward propagation process, determined by the forward propagation computational graph, is usually suboptimal. mLoRA addresses this by constructing more efficient computational graphs to reduce unnecessary overhead rather than relying on the automatically generated computational graph. For example, Figure 6's right diagram illustrates how mLoRA prunes the derivatives of the split operator within the backward propagation graph. Once the intermediate values ∇Y and all backward propagated errors ∇x_i with LoRA adapters are computed, mLoRA adds ∇x_i to the corresponding positions of ∇Y using their positional information from the forward propagation, thus generating the final result ∇X
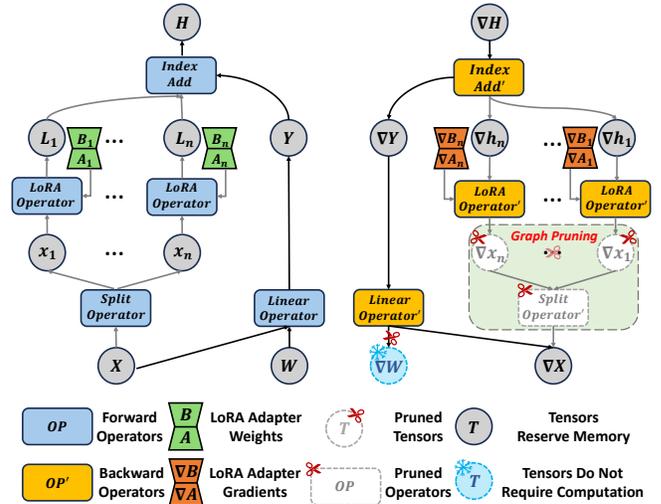

Figure 6: Computational graphs of BatchLoRA operator with graph pruning.

and avoiding expensive memory operation overhead associated with split operator derivatives.

BatchLoRA-enahnced LoRAPP. BatchLoRA complements LoRAPP to deliver highly efficient pipeline parallelism. As illustrated in Figure 4 (f), mLoRA first aims for "zero bubbles" by matching the number of fine-tuning tasks to the number of GPUs whenever possible. BatchLoRA then consolidates any additional tasks to maintain this zero-bubble condition, ensuring that the number of combined tasks equals the number of GPUs. As discussed in Section 3.4, mLoRA schedules as many tasks as the GPU memory allows, optimizing resource utilization.

3.3.2 Cost Analysis. To understand how BatchLoRA reduces overall training time for multiple fine-tuning tasks, we analyze its impact on minimizing the overhead associated with launching GPU kernel functions and the operational overhead introduced by BatchLoRA.

Kernel launch cost. As the cost of launching GPU kernel functions is proportional to the number of times the CUDA API is called [74], we define the kernel launch cost as the number of these calls. We assume that when fine-tuning one LoRA and conducting one complete forward and backward propagation, the kernel launch cost incurred by the pre-trained model's participation is α, and the kernel launch cost for each LoRA adapter is β.

When fine-tuning k LoRA adapters without using the BatchLoRA operator (Algorithm 1), each LoRA adapter and the pre-trained model conducts one complete forward and backward propagation using training data, resulting in the kernel launch cost of $k\alpha + k\beta$. When using the BatchLoRA operator (Algorithm 2), the pre-trained model conducts one complete forward and backward propagation using the consolidated data, and each LoRA adapter conducts one complete forward and backward propagation using the training data, resulting in the kernel launch cost of $\alpha + k\beta$.

Therefore, BatchLoRA can reduce the kernel launch cost by approximately $((k-1)\alpha)/(k(\alpha + \beta))$. Since LoRA adapters hold significantly fewer parameters and matrix operations compared to

the pre-trained model, it results in a much smaller cost, i.e., $\beta \ll \alpha$. Thus, the reduction in kernel launch cost is approximately $(k-1)/k$, where k is the number of concurrently trained LoRA adapters.

BatchLoRA operator cost. The split operation pruned by the BatchLoRA operator does not alter the computational workload but reduces peak memory usage during the consolidation of multiple LoRA. The memory savings equal the size of the input training data gradients, which matches the size of the input data. Assuming the total length of input tokens is O, and the hidden size of the model is h, each operator can save peak memory of $4Oh$ bytes in fp32 training precision. Moreover, it also reduces the latency associated with allocating and copying the redundant memory on GPUs.

3.4 Task Scheduler

The default scheduling objective of mLoRA is to schedule as many fine-tuning tasks as possible for high-throughput while satisfying user priorities and avoiding out-of-memory (OOM) errors [1]. This section will explain how mLoRA achieve this goal.

Preemptive priority scheduling. mLoRA uses a priority scheduling algorithm to address users' priority needs – a common practice in multi-tenant environments. Each fine-tuning task is assigned a static priority, with the highest-priority tasks processed first. Tasks with the same priority are handled on a first-come, first-served basis. Scheduling decisions are made at the end of each iteration to promptly accommodate the preemption of high-priority tasks.

Modeling memory usage. To achieve high parallelism and GPU efficiency, mLoRA schedules as many fine-tuning tasks as possible to maximize GPU memory utilization meanwhile avoiding OOM errors. To this end, mLoRA estimates the memory requirements of each fine-tuning task during task runtime. Specifically, mLoRA infers the relationship between memory size and the size of input training data as described in Vijay et al. [35]. The memory size of the attention matrix is proportional to the square of the input sequence length, while the memory size of the linear project matrix is proportional to the input sequence length. Therefore, it conducts online model fitting in the following manner:

$$Mem = \beta_0 + \beta_1 B_t L_n + \beta_2 B_t L_n^2 \qquad (5)$$

Where Mem represents the required memory; L_n is the input training data sequence length; B_t is the input batch size; β_0, β_1, and β_2 are non-negative coefficients. Throughout the model training process, mLoRA continuously gathers data points (B_t, L_n, Mem) via the profiler (Figure 2) and utilizes a non-linear least squares solver to determine the optimal coefficients for fitting this model [7]. mLoRA only needs to ensure that the total memory required by all the scheduled fine-tuning tasks is less than the available memory to avoid OOM.

4 EVALUATION

To demonstrate the effectiveness of mLoRA, we first evaluate the end-to-end performance in both single-GPU and multi-GPU environments with one or multiple machines (§ 4.2). We then examine the benefits of the LoRAPP parallelism strategy (§ 4.3) and the BatchLoRA operator (§ 4.4), respectively.

4.1 Experimental Setup

Models. We evaluate mLoRA using four publicly accessible LLaMA model series, each with different parameter scales: Llama2-70B [66], Llama2-13B, Llama2-7B, and TinyLlama-1.1B [75].

Platforms. Our experimental platforms include both single-machine and multi-machine setups. In the single-machine setup, we use four (or eight) NVIDIA RTX A6000 GPUs, each with 48GB of memory, connected via PCIe 4.0x16. For the multi-machine setup, we utilize eight NVIDIA GeForce RTX 3090 GPUs, each with 24GB of memory, distributed across eight machines connected through 1Gbps networking [2]. Each machine is equipped with an Intel Xeon Silver 4314 CPU and 256GB of RAM. In the single-machine setup, we further distinguish between the single-GPU mode, using one RTX A6000 GPU, and the single-machine, multi-GPU mode, which defaults to four RTX A6000 GPUs unless specified otherwise. For the multi-machine setup, the default configuration is the multi-machine, multi-GPU mode with eight RTX 3090 GPUs. We use eight NVIDIA RTX A6000 GPUs to test mLoRA's scalability.

Workloads. In all experiments, we use the natural language generation (NLG) dataset GSM8K [21] to evaluate the performance of the training systems. Following the default hyperparameter settings of Alpaca-LoRA [1], we fine-tune the PLMs with a batch size of 8, a sequence length of 512, 10 epochs, and a LoRA adapter rank of 16. The LoRA adapter is applied to the linear layers of the PLMs, i.e., q_proj, k_proj, v_proj, and o_proj.

Performance Metrics. We report the average fine-tuning task completion time, which is the average time required to complete a fine-tuning task, and the system throughput, defined as the total number of tokens the system can train per second.

Baselines. In the single-GPU environment, we compare mLoRA with HuggingFace **PEFT** [48], the state-of-the-art library for training parameter-efficient fine-tuning models. Due to memory constraints, it is not feasible to use 32fp precision to fine-tune PLMs in this setup (unlike in a multi-GPU setup), so we use 8-bit quantization and activation checkpointing techniques for both mLoRA and PEFT to reduce memory overhead.

In the multi-GPU environments, whether for single-machine or multiple-machine setups, we compare mLoRA with three state-of-the-art model parallelism strategies: 1) One Forward Pass followed by One Backward Pass (**1F1B**), a synchronous gradient update pipeline parallelism similar to GPipe but more memory-efficient, introduced by PipeDream-Flush [52]. 2) Tensor Parallelism for Transformers (**TP**), an optimized model parallelism method for the transformer architecture proposed by Megatron-LM [53]; 3) Fully Sharded Data Parallel [79] (**FSDP**), an industry-grade parallel LLM training strategy which combines the data and model parallelism and employs the Zero Redundancy Optimizer [57, 60] technology proposed by DeepSpeed [58]. Note that training LoRA models on multiple GPUs without model parallelism – where each GPU holds a complete copy of the base model and trains separate LoRA models – is impractical in our evaluation due to significant memory limitations. As a result, we exclude the data parallelism strategy from our multi-GPU environment comparisons.
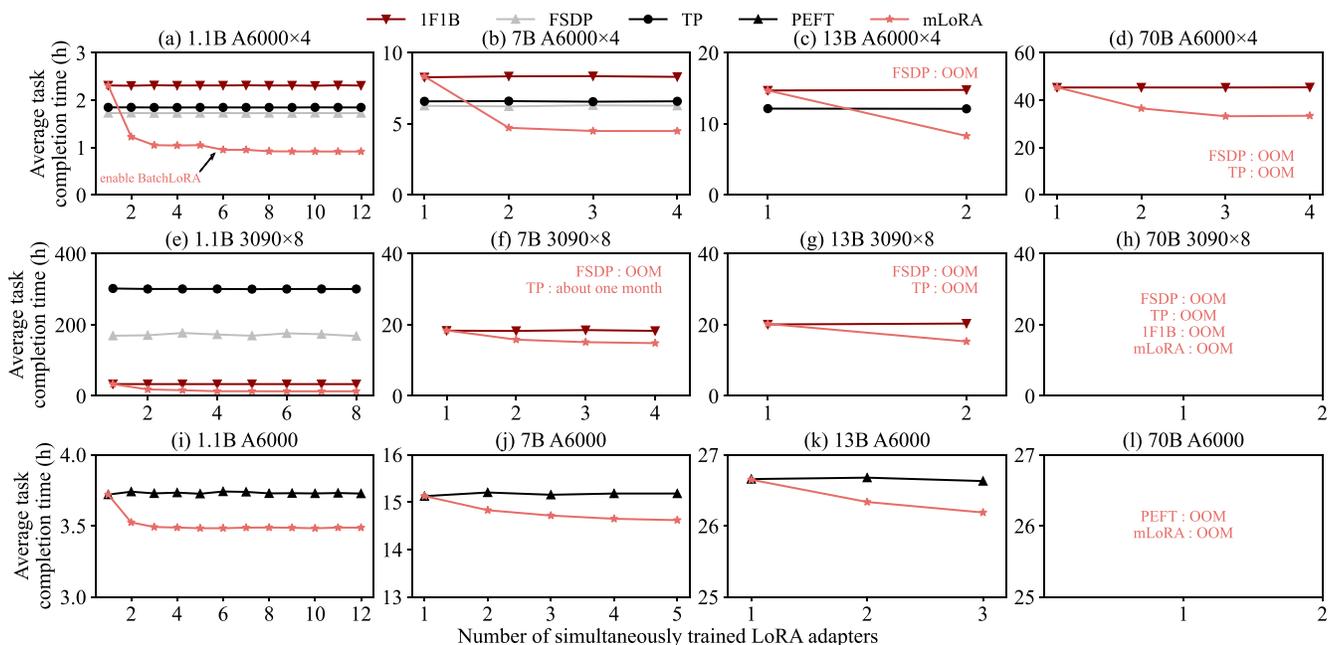
Figure 7: (a), (b), (c), and (d) show the average fine-tuning task completion time in the single-machine multi-GPU setup. (e), (f), (g), and (h) show the average fine-tuning task completion time in the multi-machine multi-GPU. (i), (j), (k), and (l) show the average fine-tuning task completion time in the single-GPU setup. In the single-GPU setup, we can only run mLoRA using BatchLoRA; In the multi-GPU setup, we run mLoRA using BatchLoRA and LoRAPP. For tine-tuning the 70B model in both single-GPU and multi-GPU setups, we use 8-bit quantization and activation checkpointing techniques in all baseline and mLoRA, as the model cannot fit in 32fp precision even in the multi-GPU setup. Note that, we enable BatachLoRA only when the number of fine-tuning tasks surpasses the number of GPUs to ensure zero pipeline bubbles.

4.2 End-to-End Results

In this section, we present the end-to-end performance results between mLoRA and the state-of-the-art. As the number of simultaneous fine-tuning tasks affects mLoRA's performance, we gradually increase the number of simultaneous fine-tuning tasks until the system's memory capacity is reached. Each task maintains the parameter settings as outlined in Section 4.1, with only modifications to hyperparameters unrelated to throughput, such as learning rate.

Results in single-machine, multi-GPU mode: As shown in Figure 7 (a), (b), and (c), mLoRA achieves an average task completion time reduction of 30% to 45% in single-machine, multi-GPU mode with three models of varying parameter scales, thanks to LoRAPP, which decreases communication latency compared to state-of-the-art methods such as TP, and FSDP. Notably, FSDP suffers from additional memory overhead due to parameter replication, preventing it from training a 13B model in a single-machine, multi-GPU mode. In contrast, LoRAPP enables mLoRA to train up to two 13B models simultaneously, as shown in Figure 7 (c).

Results in multi-machine, multi-GPU mode: In the multi-machine, multi-GPU mode, as shown in Figure 7 (e), (f) and (g), it is merely possible for FSDP and TP to train relatively large models given a low-bandwidth cluster (e.g., 1Gbps in our setup). Although the total GPU memory is the same as that in single-machine, multi-GPU mode, FSDP, and TP incur additional memory overhead on

each node, making FSDP impossible to train a 7B or a 13B model and for TP to train a 13B model. In contrast, compared to 1F1B, mLoRA saves 30% in average task completion time for the 7B model, as shown in Figure 7 (f), due to LoRAPP reducing pipeline bubbles. As communication becomes a bottleneck in this setup, resulting in nearly identical training times for both 7B and 13B models.

Results in single GPU: In the single-GPU setup, as shown in Figure 7 (i), (j), and (k), mLoRA reduces the average task completion time by up to 8% due to the BatchLoRA operator, which decreases the overhead of launching kernel functions. We note that as the base model size increases, the overhead of launching kernel functions constitutes a smaller proportion, resulting in reduced performance gains by BatchLoRA (e.g., a 2% reduction with a 13B model).

Results in large model: The experimental results, as shown in Figure 7 (l), show that even with 8-bit quantization and activation checkpointing techniques applied across all baseline methods and mLoRA, fine-tuning such a large model remains infeasible in a single GPU setup due to memory limitations. This highlights the necessity of parallel fine-tuning large models in a multi-GPU environment. Moreover, as shown in Figure 7 (h), despite the larger overall memory capacity in the multi-machine, multi-GPU mode, the smaller per-GPU memory (which limits peak memory) still makes fine-tuning such large models infeasible. Therefore, we could only fine-tune such models in a single-machine, multi-GPU mode with
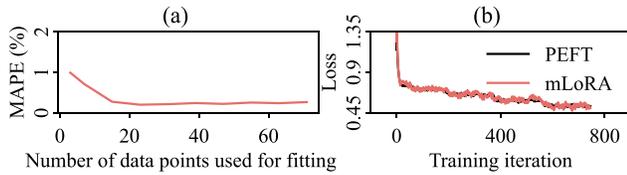
Figure 8: (a) The accuracy of the online model fitting. (b) Model training convergence study.
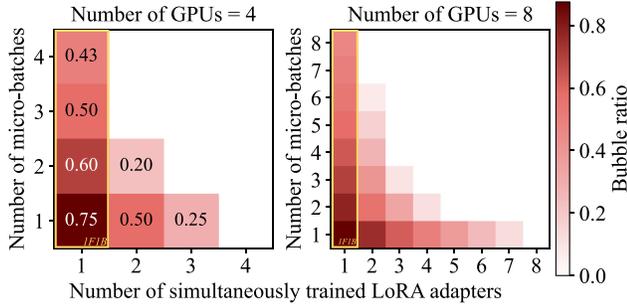

Figure 9: Bubble ratio of LoRAPP and 1F1B.

GPUs that have larger memory. The result, as shown in Figure 7 (d), indicates that mLoRA reduces the average task completion time by 30% compared to 1F1B. Furthermore, due to the additional memory overhead, neither FSDP nor TP can fine-tune models of this scale.

Moreover, as introduced in Section 3.3, we can further enhance performance using BatchLoRA when the pipeline reaches zero bubble state (e.g., with more than 4 fine-tuning tasks on 4 GPUs according to Section 3.2.2). As shown in Figure 7 (a), when the number of simultaneous training tasks reaches 6, mLoRA enables the BatchLoRA operator, resulting in an additional 10% performance improvement, saving 40% in average task completion time.

It is important to note that the performance of all baseline methods remains unchanged as the number of LoRA adapters increases. This is because these methods are designed to fine-tune only a single LoRA adapter. When multiple LoRA adapters need to be fine-tuned, the baseline methods must execute them sequentially, as outlined in Algorithm 1, resulting in no improvement in throughput.

Model convergence. We track the loss values for each LoRA adapter during training, as shown in Figure 8 (b). mLoRA exhibits a convergence trend similar to PEFT, indicating that mLoRA achieves the same performance as PEFT.

Memory usage modeling. We evaluate the accuracy of the online model fitting used in mLoRA's scheduler (§3.4) for predicting the memory usage of each fine-tuning task. Since the number of data points used to fit the model affects its prediction mean absolute percentage error (MAPE), we use different data points to test its prediction MAPE, as shown in Figure 8 (a). The results show that the model achieves a high accuracy, approximately 0.25% MAPE, even with a limited number of data points. Practically, we can set an error margin of 0.25% for GPU memory usage estimates to avoid out-of-memory (OOM) errors.
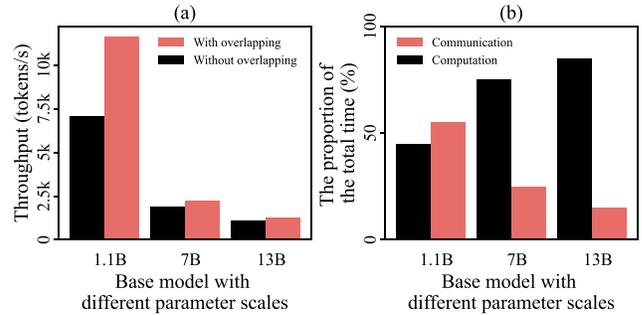

Figure 10: (a) Comparison of throughput with and without overlapping communication. (b) The proportion of computation and communication time in the total fine-tuning time.

4.3 Effectiveness of LoRAPP

In this section, we focus on evaluating the impact of LoRAPP (i.e., no LoRABatch) in single-machine, multi-GPU mode. First, we examine the performance differences between LoRAPP and 1F1B, focusing on bubble ratio. Next, we analyze the efficiency of overlapping communication. We then compare LoRAPP with 1F1B, TP, and FSDP in terms of throughput. Finally, we assess the scalability and adaptability of mLoRA's LoRAPP parallelism strategy.

Bubble ratio analysis. Given that the LoRAPP parallelism method also incorporates a mechanism akin to 1F1B, i.e., dividing mini-batch data into micro-batches to reduce the bubble ratio, we analyze the correlation between the bubble ratio, the number of micro-batches, and the number of simultaneously trained LoRA adapters for both mLoRA and 1F1B. The results, as depicted in Figure 9, show that when fine-tuning a single LoRA adapter, the bubble ratio of mLoRA is comparable to that of 1F1B, i.e., the bubble ratio of 1F1B is a special case of using mLoRA to fine-tune a single LoRA adapter. The bubble ratio of 1F1B decreases gradually with an increasing number of micro-batches but never reaches zero. In contrast, mLoRA can rapidly reduce the bubble ratio to zero by increasing the number of simultaneously trained LoRA adapters.

Effectiveness of overlapping communication. Figure 10 (a) shows that overlapping communication increases the throughput by 60% for the 1.1B model, 20% for the 7B model, and 16% for the 13B model compared to the case without overlapping communication. The 1.1B model benefits more from communication overlap, as shown in Figure 10 (b), due to its higher proportion of communication relative to the overall fine-tuning time, allowing more communication time to be hidden. In contrast, the 7B and 13B models benefit less due to their lower communication proportion.

Performace. We first present the highest throughput (i.e., tokens per second) achieved by each approach in Figure 11. Due to LoRAPP's smaller communication volume compared to TP (and FSDP) and lower bubble ratio than 1F1B, mLoRA exhibits superior performance. For the 1.1B model, mLoRA's throughput is 75% higher than FSDP and 86% higher than TP. For the 7B model, mLoRA outperforms FSDP by 35% and TP by 58%. For the 13B model, FSDP encounters an OOM error due to the need for additional memory to store weight copies exceeding GPU capacity, while mLoRA achieves
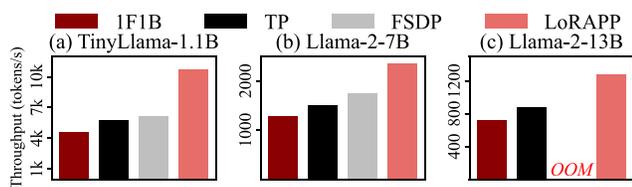
Figure 11: Throughput comparisons among different parallelism strategies.
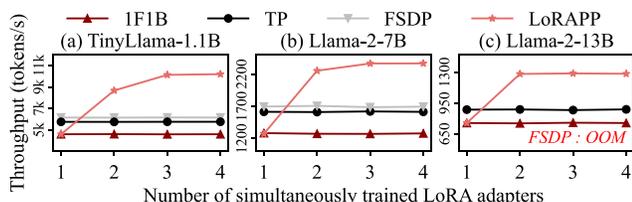
Figure 12: Throughput comparisons among different parallelism strategies with varying numbers of simultaneously trained LoRA adapters.
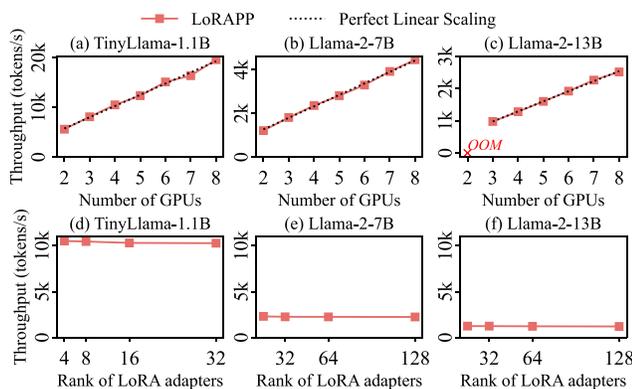
Figure 13: (a), (b), and (c) show the linear scalability of LoRAPP. (d), (e), and (f) show the adaptability of mLoRA.

a throughput 46% higher than TP. As LoRAPP greatly reduces communication overhead, mLoRA benefits the 1.1B model more, which has lower computational overhead and higher communication costs relative to overall training time. In contrast, the 13B model benefits less due to much higher computational overhead.

We then compare throughput by varying the number of simultaneously trained LoRA adapters. As shown in Figure 12, mLoRA's throughput increases with the number of simultaneously trained LoRA adapters until a bubble ratio of zero is achieved (i.e., four LoRA adapters trained simultaneously in a four-GPU setup). Beyond this point, no further improvements are observed, as BatchLoRA is not enabled. In contrast, the throughput of 1F1B, TP, and FSDP remains constant regardless of the number of simultaneously trained LoRA adapters. Specifically, mLoRA outperforms 1F1B in throughput due to its ability to achieve a smaller bubble ratio. Additionally, when

only a single LoRA adapter is trained, the higher bubble ratio results in lower throughput for both mLoRA and 1F1B. Furthermore, because mLoRA has a lower communication volume compared to TP and FSDP, it surpasses these strategies in throughput when training more than one LoRA adapter. Note that while launching multiple instances of 1F1B, TP, or FSDP on these GPUs to train multiple adapters could increase throughput, it quickly consumes additional memory and may trigger OOM errors.

Scalability. To evaluate the scalability of mLoRA, we train LoRA models using an increasing number of GPUs, ranging from 2 to 8. The throughput is recorded when the number of simultaneously trained LoRA adapters reaches its maximum. The results, as shown in Figure 13 (a), (b), and (c), indicate that mLoRA's throughput increases linearly with the number of GPUs.

Adaptability. To evaluate the adaptability of mLoRA across different model scales, we train LoRA models with varying LoRA adapter ranks in the single-machine, multi-GPU mode. The throughput is recorded when the number of simultaneously trained LoRA adapters reaches its maximum. The results, as shown in Figure 13 (d), (e), and (f), show that since the rank remains low (i.e., keep the rank much smaller than the base model's hidden size), different ranks have almost no impact on performance. Therefore, mLoRA remains effective when handling LoRA adapters with different ranks.

4.4 Effectiveness of BatchLoRA

In this section, we examine the impact of the BatchLoRA operator in the single-GPU setup. Given the orthogonal nature of the BatchLoRA operator and the LoRAPP parallelism, the results remain consistent as those in the multi-GPU setup (§ 4.2).

To understand how BatchLoRA mitigates the overhead of kernel function launches, we employ NVIDIA's performance analysis tool, NVIDIA Nsight Systems [5], to monitor kernel launch times and kernel execution time. Recall that the effectiveness of the BatchLoRA operator is affected by the number of simultaneously trained LoRA adapters, as it can reduce the number of kernel function launches for multiple tasks to the same level as for a single task. Therefore, we increase the number of simultaneously training LoRA adapters and measure the corresponding kernel launch times and kernel execution times. When the number of simultaneously trained adapters reaches its maximum, we compare the training time, proportion of kernel launch time, and kernel execution time per fine-tuning task at each training step. In addition, to evaluate the effectiveness of mLoRA's graph pruning approach (§ 3.3.1) in optimizing the computation graph, we record the forward and backward propagation time and the peak GPU memory consumption.

Figure 14 (a) shows that mLoRA reduces the training time by 8% for the 1.1B model, 5% for the 7B model, and 2% for the 13B model, compared to HuggingFace PEFT. This improvement is due to the fact that, as illustrated in Figure 14 (b), the overhead from launching kernel functions accounts for 10% of the total overhead for the 1.1B model, and optimizing this aspect leads to significant time savings. In contrast, for the 7B model, the overhead is 7.5%, and for the 13B model, it is 4%, resulting in smaller reductions in training time. As shown in Figure 14 (c), since the computational workload of the BatchLoRA operator is comparable to that of PEFT, there is almost no difference in kernel execution time.
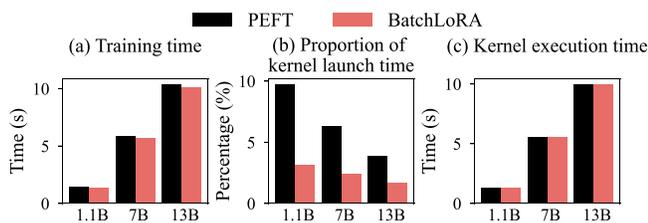
Figure 14: The comparisons of training time, proportion of kernel launch, and kernel execution time between PEFT and BatchLoRA per fine-tuning task at each training step.
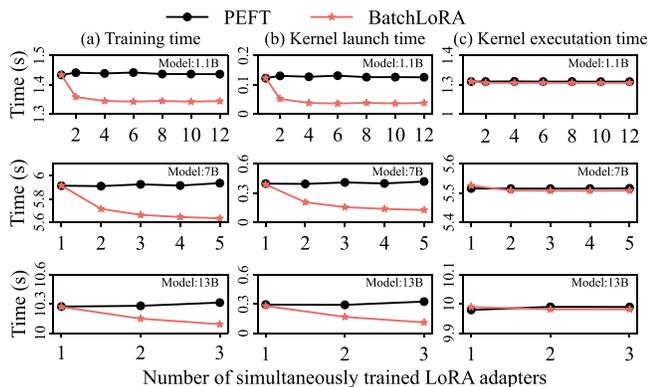

Figure 15: The training time, kernel launch time, and kernel execution time for per fine-tuning task at each training step.
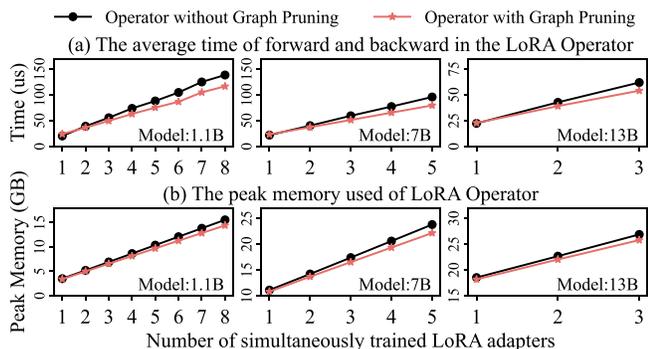

Figure 16: Performance comparisons between the BatchLoRA operator implemented by PyTorch and the operator with graph pruning.

Figure 15 (a) shows that due to PEFT's inability to batch multiple fine-tuning tasks, the average latency for each training step/iteration remains unchanged. In contrast, with BatchLoRA, the average latency per training step decreases (i.e., becomes more efficient) as the number of simultaneously trained LoRA adapters increases. This is because, as shown in Figure 15 (b) and (c), while the kernel execution time remains nearly identical for both BatchLoRA

and PEFT as the number of simultaneously trained LoRA adapters increases, the BatchLoRA operator reduces kernel launch time.

Figure 16 shows that mLoRA, through its graph pruning (§ 3.3.1), outperforms the operator without graph pruning in both latency and peak memory usage when simultaneously training multiple fine-tuning tasks. Additionally, mLoRA achieves up to a 17% reduction in latency and a 7% reduction in peak memory usage as the number of simultaneous fine-tuning tasks increases.

5 RELATED WORK

Parameter-efficient fine-tuning. While mLoRA focuses on a typical LoRA implementation due to its wide adoption, most LoRA-based fine-tuning methods [23, 39, 46, 76] can also be easily applied to mLoRA, as they all follow the same scheme – i.e., one base model is associated with adapters.

LoRA-based multi-task systems. Other works, such as Punica [17] and S-LoRA [62], explored the potential of serving multi-task inference services by sharing base-model weights with batched inference requests from different LoRA adapters. However, those optimizations cannot be directly applied to the training process as they do not address operators generated by backward.

Pipeline mechanisms for model training. Pipelining has been leveraged to improve the performance of machine learning systems [16, 19, 32, 44, 51]. Pipelined back propagation [19] handles the expensive back propagation. Pipe-SGD pipelines the processing of a minibatch to hide communication time in AllReduce-based systems [44]. A weight prediction technique is proposed to address the staleness issue in pipelined model parallelism [16]. PipeDream [51] employs the one-forward-one-backward scheduling algorithm for pipeline execution where the minimum number of mini-batches that is large enough to saturate the pipeline is admitted. mLoRA's LoRAPP, specifically targeting LoRA-based fine-tuning, is orthogonal to these optimizations.

6 CONCLUSION

We have presented mLoRA, a fine-tuning system designed and developed for efficiently training multiple LoRA adapters across GPUs and machines. The proposed techniques, including LoRA-aware streamlined pipeline parallelism and a LoRA-efficient training operator, allow mLoRA to fully utilize the computational and memory capacities of a multiple-GPU training cluster. Our extensive evaluation demonstrates that mLoRA significantly reduces average fine-tuning time and improves training throughput compared to state-of-the-art methods. Deployed in a production environment at AntGroup, mLoRA has achieved over 30% time savings in selecting optimal hyperparameters for fine-tuning LLM models. Moreover, mLoRA facilitates efficient multi-LoRA fine-tuning on cost-effective GPUs, making LLMs more accessible.

REFERENCES

[1] 2023. Alpaca-LoRA. https://github.com/tloen/alpaca-lora.
[2] 2024. Announcing Anyscale Private Endpoints and Anyscale Endpoints Fine-tuning. https://www.anyscale.com/.
[3] 2024. Customize a model with azure open AI service. https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/fine-tuning.
[4] 2024. Hugging Face. https://huggingface.co/.
[5] 2024. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.
[6] 2024. OpenAI fine-tuning. https://platform.openai.com/docs/guides/fine-tuning.
[7] 2024. SciPy: Solve a nonlinear least-squares problem with bounds on the variables. https://docs.scipy.org/doc/scipy-1.13.0/reference/generated/scipy.optimize.least_squares.html.
[8] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, and Arash Bakhtiari et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL]
[9] Alan Ansell, Edoardo Maria Ponti, Anna Korhonen, and Ivan Vulić. 2021. Composable sparse fine-tuning for cross-lingual transfer. arXiv preprint arXiv:2110.07560 (2021).
[10] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. Proceedings of the VLDB Endowment 17, 2 (2023), 92–105.
[11] Akari Asai, Mohammadreza Salehi, Matthew E Peters, and Hannaneh Hajishirzi. 2022. ATTEMPT: Parameter-efficient multi-task tuning via attentional mixtures of soft prompts. arXiv preprint arXiv:2205.11961 (2022).
[12] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. 2023. vTrain: A Simulation Framework for Evaluating Cost-effective and Compute-optimal Large Language Model Training. arXiv preprint arXiv:2312.12391 (2023).
[13] Yakoub Bazi, Laila Bashmal, Mohamad Mahmoud Al Rahhal, Riccardo Ricci, and Farid Melgani. 2024. RS-LLaVA: A Large Vision-Language Model for Joint Captioning and Question Answering in Remote Sensing Imagery. Remote Sensing 16, 9 (2024). https://doi.org/10.3390/rs16091477
[14] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys (CSUR) 52, 4 (2019), 1–43.
[15] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258 (2021).
[16] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. arXiv preprint arXiv:1809.02839 (2018).
[17] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2023. Punica: Multi-Tenant LoRA Serving. CoRR abs/2310.18547 (2023). https://doi.org/10.48550/ARXIV.2310.18547 arXiv:2310.18547
[18] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016).
[19] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. 2012. Pipelined Back-Propagation for Context-Dependent Deep Neural Networks.. In Interspeech. 26–29.
[20] Daning Cheng, Shigang Li, Hanping Zhang, Fen Xia, and Yunquan Zhang. 2021. Why dataset properties bound the scalability of parallel machine learning training algorithms. IEEE Transactions on Parallel and Distributed Systems 32, 7 (2021), 1702–1712.
[21] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. arXiv preprint arXiv:2110.14168 (2021).
[22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. arXiv:2208.07339 [cs.LG]
[23] Ning Ding, Xingtai Lv, Qiaosen Wang, Yulin Chen, Bowen Zhou, Zhiyuan Liu, and Maosong Sun. 2023. Sparse low-rank adaptation of pre-trained language models. arXiv preprint arXiv:2311.11696 (2023).
[24] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. Nature Machine Intelligence 5, 3 (2023), 220–235.
[25] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 431–445.
[26] Benjamin Feuer, Yurong Liu, Chinmay Hegde, and Juliana Freire. 2024. ArcheType: A Novel Framework for Open-Source Column Type Annotation using Large Language Models. Proc. VLDB Endow. 17, 9 (2024), 2279–2292. https://www.vldb.org/pvldb/vol17/p2279-freire.pdf

[27] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. Proceedings of the VLDB Endowment 17, 5 (2024), 1132–1145.
[28] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. arXiv:2403.14608 [cs.LG]
[29] Yuchao Li Donglin Zhuang Zhongzhu Zhou Xiafei Qiu Yong Li Wei Lin Shuaiwen Leon Song Haojun Xia, Zhen Zheng. 2024. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. Proceedings of the VLDB Endowment 17, 2 (2024), 211–224.
[30] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In International conference on machine learning. PMLR, 2790–2799.
[31] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net. https://openreview.net/forum?id=nZeVKeeFYf9
[32] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems 32 (2019).
[33] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. Proceedings of Machine Learning and Systems 1 (2019), 1–13.
[34] Ming Jin, Shiyu Wang, Lintao Ma, Zhixuan Chu, James Y. Zhang, Xiaoming Shi, Pin-Yu Chen, Yuxuan Liang, Yuan-Fang Li, Shirui Pan, and Qingsong Wen. 2023. Time-LLM: Time Series Forecasting by Reprogramming Large Language Models. CoRR abs/2310.01728 (2023). https://doi.org/10.48550/ARXIV.2310.01728 arXiv:2310.01728
[35] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. Proceedings of Machine Learning and Systems 5 (2023).
[36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles. 611–626.
[37] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. Proceedings of the VLDB Endowment 17, 8 (2024), 1939–1952.
[38] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. arXiv preprint arXiv:2104.08691 (2021).
[39] Dengchun Li, Yingzi Ma, Naizheng Wang, Zhiyuan Cheng, Lei Duan, Jie Zuo, Cal Yang, and Mingjie Tang. 2024. MixLoRA: Enhancing Large Language Models Fine-Tuning with LoRA based Mixture of Experts. arXiv preprint arXiv:2404.15159 (2024).
[40] Ding Li and Zhang Xian. 2023. TianPeng: A Chinese chat model that is fine-tuned using LoRA on top of the LLaMA-30B model. https://huggingface.co/pleisto/tianpeng-lora-30b. https://doi.org/10.57967/hf/0528
[41] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 661–670.
[42] Shigang Li, Tal Ben-Nun, Giorgi Nadiradze, Salvatore Di Girolamo, Nikoli Dryden, Dan Alistarh, and Torsten Hoefler. 2020. Breaking (global) barriers in parallel stochastic optimization with wait-avoiding group averaging. IEEE Transactions on Parallel and Distributed Systems 32, 7 (2020), 1725–1739.
[43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. [n. d.]. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. Proceedings of the VLDB Endowment 13, 12 ([n. d.]).
[44] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. 2018. Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training. Advances in Neural Information Processing Systems 31 (2018).
[45] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous decentralized parallel stochastic gradient descent. In International Conference on Machine Learning. PMLR, 3043–3052.
[46] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. 2024. DoRA: Weight-Decomposed Low-Rank Adaptation. CoRR abs/2402.09353 (2024). https://doi.org/10.48550/ARXIV.2402.09353 arXiv:2402.09353
[47] Gen Luo, Yiyi Zhou, Tianhe Ren, Shengxin Chen, Xiaoshuai Sun, and Rongrong Ji. 2024. Cheap and quick: Efficient vision-language instruction tuning for large language models. Advances in Neural Information Processing Systems 36 (2024).

[48] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.

[49] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. *arXiv preprint arXiv:2402.06196* (2024).

[50] Giorgi Nadiradze, Amirmojtaba Sabour, Dan Alistarh, Aditya Sharma, Ilia Markov, and Vitaly Aksenov. 2019. SwarmSGD: Scalable decentralized SGD with local updates. *arXiv preprint arXiv:1910.12308* (2019).

[51] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[52] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[53] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[54] Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. 2024. Unifying large language models and knowledge graphs: A roadmap. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[56] Rajvardhan Patil and Venkat Gudivada. 2024. A Review of Current Trends, Techniques, and Challenges in Large Language Models (LLMs). *Applied Sciences* 14, 5 (2024), 2074.

[57] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[58] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[59] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. 2017. Learning multiple visual domains with residual adapters. *Advances in neural information processing systems* 30 (2017).

[60] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[61] Yikang Shen, Zhen Guo, Tianle Cai, and Zengyi Qin. 2024. JetMoE: Reaching Llama2 Performance with 0.1 M Dollars. *arXiv preprint arXiv:2404.07413* (2024).

[62] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. 2023. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285* (2023).

[63] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. 2023. Fairness in serving large language models. *arXiv preprint arXiv:2401.00588* (2023).

[64] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[65] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

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

[67] C Tribes, S Benarroch-Lelong, P Lu, and I Kobyzev. 2023. Hyperparameter optimization for Large Language Model instruction-tuning. *Les Cahiers du GERAD ISSN* 711 (2023), 2440.

[68] Ben Wodecki. 2024. AI's New Frontier: Training Trillion-Parameter Models with Much Fewer GPUs. *https://aibusiness.com/nlp/ai-s-new-frontier-training-trillion-parameter-models* (2024).

[69] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. arXiv:2305.05920 [cs.LG]

[70] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 911–927. https://www.usenix.org/conference/osdi24/presentation/wu-bingyang

[71] Ming Xu. 2023. pycorrector: Text Error Correction Tool. https://github.com/shibing624/pycorrector

[72] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. 2021. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems* 3 (2021), 269–296.

[73] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[74] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. 2019. Understanding the overheads of launching CUDA kernels. *ICPP19* (2019), 5–8.

[75] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. arXiv:2401.02385 [cs.CL]

[76] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2023. Adaptive budget allocation for parameter-efficient fine-tuning. In *The Eleventh International Conference on Learning Representations*.

[77] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. 2024. LoRA Land: 310 Fine-tuned LLMs that Rival GPT-4, A Technical Report. *arXiv preprint arXiv:2405.00732* (2024).

[78] Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. 2020. Masking as an efficient alternative to finetuning for pretrained language models. *arXiv preprint arXiv:2004.12406* (2020).

[79] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3848–3860.