FlexLLM: A System for Co-Serving Large Language Model Inference and Parameter-Efficient Finetuning

Xupeng Miao* Gabriele Oliaro* Xinhao Cheng Mengdi Wu Colin Unger† Zhihao Jia

Carnegie Mellon University *Stanford University*†

Abstract

Parameter-efficient finetuning (PEFT) is a widely used technique to adapt large language models for different tasks. Service providers typically create separate systems for users to perform PEFT model finetuning and inference tasks. This is because existing systems cannot handle workloads that include a mix of inference and PEFT finetuning requests. As a result, shared GPU resources are underutilized, leading to inefficiencies. To address this problem, we present FlexLLM, the first system that can serve inference and parameter-efficient finetuning requests in the same iteration. Our system leverages the complementary nature of these two tasks and utilizes shared GPU resources to run them jointly, using a method called *co-serving*. To achieve this, FlexLLM introduces a novel *token-level finetuning* mechanism, which breaks down the finetuning computation of a sequence into smaller token-level computations and uses *dependent parallelization* and *graph pruning*, two static compilation optimizations, to minimize the memory overhead and latency for co-serving. Compared to existing systems, FlexLLM's co-serving approach reduces the activation GPU memory overhead by up to 8×, and the end-to-end GPU memory requirement of finetuning by up to 36% while maintaining a low inference latency and improving finetuning throughput. For example, under a heavy inference workload, FlexLLM can still preserve more than 80% of the peak finetuning throughput, whereas existing systems cannot make any progress with finetuning. The source code of FlexLLM is publicly available at https://github.com/flexflow/FlexFlow/.

1 Introduction

Recent advancements in generative large language models (LLMs) such as GPT [9,22,47], PaLM [14], and LLaMA [62] have shown strong capabilities of generating natural language texts across various application domains [21,39,60,72]. The ongoing evolution of LLMs is accompanied by exponential increases in their model sizes, with some models encompassing more than 100 billion parameters [50,53], making it infeasible to fully finetune an LLM for downstream tasks (i.e., training on task-specific datasets).

To reduce the computational requirement of LLM finetuning, recent work introduces *parameter-efficient finetuning* (PEFT), which updates a subset of trainable parameters from an LLM or introduces a small number of new parameters into the LLM while keeping the vast majority of the original LLM parameters frozen [26, 28, 29, 34, 36, 48]. As a result, PEFT only requires maintaining a small fraction of optimizer states for updating model parameters, dramatically reducing the memory requirement compared to full finetuning [17]. The recent progress of PEFT-based LLMs has proved that PEFT can achieve comparable performance to full finetuning while enabling fast adaption to new tasks [48].

Given the distinct needs of training and inference tasks, which are respectively optimized for throughput and latency, existing cluster schedulers generally deploy the two tasks separately [11, 23]. More importantly, current LLM runtime systems are designed to support either PEFT-based LLM finetuning task *or* inference tasks exclusively, but none can accommodate the concurrent serving of both task types [10, 41, 57]. As a result, researchers and practitioners have to allocate distinct GPU resources to LLM inference and finetuning tasks. This approach achieves suboptimal GPU utilization for both tasks, as shown in Figure 1. In particular, due to the autoregressive nature of LLMs, existing LLM inference systems use *incremental decoding* to perform inference computation where the system iteratively decodes *one* token using the input prompt and all previously generated tokens. Consequently, LLM inference is primarily bounded by memory access since the decoding of each token requires accessing all model parameters [32, 43]. On the other hand, finetuning an LLM can process *all* tokens of a request simultaneously (e.g., 4K tokens for LLaMA-2 [63]), making finetuning largely bottlenecked by compute resources such as the compute capability of tensor cores on modern GPUs.

Based on a critical observation that LLM inference and finetuning tasks are *complementary*, this paper introduces

Inference 因为每个 token 都要访问全部 model param 所以 bound 在 memroy access

Finetune 很大程度上 bound 在计算上? 也需要访问全部的 model param，但可以同时处理全部 tokens

*Equal contribution.

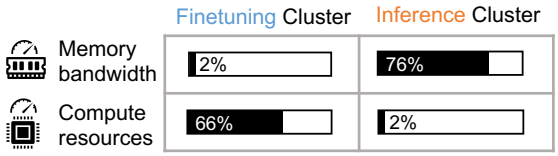

Figure 1: Average GPU memory bandwidth and compute resources utilization for LLM finetuning and inference. The results are collected by serving and finetuning a LoRA-assisted LLaMA-2-70B model on four and eight NVIDIA A100 40GB GPUs. We use a maximum sequence length of 32K for inference and a sequence length of 4K for LoRA finetuning.

FlexLLM, the first co-serving system for LLM inference and parameter-efficient finetuning[1]. FlexLLM introduces a *PEFT-as-a-service (PaaS)* interface that unifies inference and finetuning tasks and jointly serves them on shared GPU resources. This co-serving approach allows FlexLLM to achieve the best of both worlds. Figure 2 shows the inference and finetuning performance achieved by FlexLLM through co-serving. By fusing the computation of inference and finetuning and allowing them to share model weights, FlexLLM outperforms the conventional approach of employing twice as many GPUs for resource isolation. Specifically, FlexLLM retains more than 80% of the peak throughput of finetuning tasks while simultaneously preserving the latency of inference tasks. Co-serving inference and finetuning requires addressing two key challenges. Next, we elaborate on these challenges and the main ideas FlexLLM uses to manage them.

Inference latency. Most LLM inference tasks are online and dynamically fluctuating, necessitating a system that can rapidly adjust GPU resources to guarantee *low latency*. The inference of generative LLMs follows a token-by-token pattern, while a finetuning task processes an entire sequence in each iteration to maximize throughput. As a result, directly fusing the computation of an inference iteration with that of a finetuning iteration would significantly prolong the inference latency. For example, under the setting from Figure 1, an inference and a finetuning iteration take, respectively, 50ms and 1500ms.

To address this challenge, FlexLLM introduces a novel *token-level finetuning* mechanism, which decomposes the finetuning computation of a sequence into a finer granularity of multiple tokens (Section 6.1). This approach significantly reduces the per-iteration latency of finetuning for both forward and backward passes, and allows FlexLLM to finetune an arbitrary number of tokens for each iteration. FlexLLM's *hybrid token scheduler* coordinates inference and finetuning tokens for each co-serving iteration to maximize resource utilization while attaining the service level objective (SLO) for inference requests (Section 6.2).

[1]For simplicity, we use the term finetuning to refer to parameter-efficient finetuning for the rest of the paper.

Memory overhead. Within existing finetuning systems, the update process for a small group of model parameters necessitates the retention of all intermediate activations to facilitate the gradient computation of those parameters subject to training. The activations substantially contribute to the memory requirements during finetuning, making it more resource-intensive than inference. FlexLLM introduces fundamental techniques to minimize the memory requirement of PEFT. First, different from fully finetuning an LLM's weights, PEFT methods generally keep *all* weights of the original LLM *frozen* and introduce multiple *bypass networks* with small amounts of trainable parameters. Based on this observation, FlexLLM makes all finetuning and inference tasks built on top of the same base LLM so that they can share model weights and intermediate activations. FlexLLM introduces *dependent parallelization* to discover an optimized parallelization strategy for a PEFT model by solving a constrained optimization problem (Section 5.1). Instead of preserving all intermediate activations during a forward pass for finetuning as existing systems, FlexLLM leverages a critical observation that most intermediate activations are used for computing the gradients for *frozen parameters* and performs *graph pruning* to eliminate data dependencies for frozen parameters (Section 5.2). These memory optimizations allow FlexLLM to reduce PEFT memory overhead by up to 8× compared to existing systems.

Evaluation. We compare FlexLLM's co-serving approach with existing approaches that employ separate GPUs to perform inference and finetuning and show that FlexLLM achieves significantly higher finetuning throughput. For example, under heavy inference workloads where existing systems must dedicate all available GPUs for inference tasks and therefore cannot make any finetuning progress, FlexLLM can still preserve more than 80% of peak finetuning throughput.

Contributions. This paper makes the following contributions:

- We show that LLM inference and finetuning are complementary and present their first co-serving system.
- We propose a novel mechanism to decompose finetuning computation into the token-level granularity, which allows fusing inference and finetuning computation.
- We introduce dependent parallelization and graph pruning, two static compilation optimizations that reduce PEFT memory overhead by 8×.
- We build FlexLLM, a co-serving system that implements these techniques and vastly outperforms existing systems.

2 Background and Challenges

2.1 Parameter-Efficient Finetuning

Finetuning allows an LLM to adapt to specialized domains and tasks [9, 18, 49]. However, finetuning an entire LLM in-

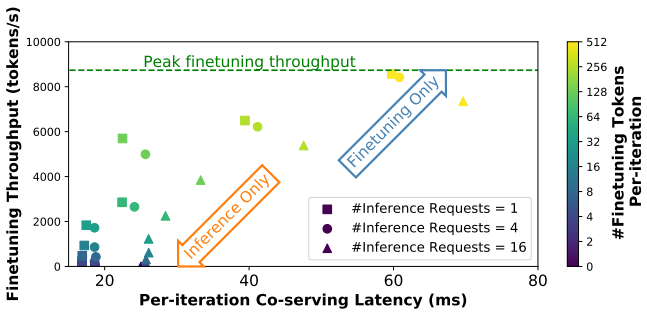

Figure 2: Co-serving LLAMA-2-70B inference and finetuning on four NVIDIA A100 40GB GPUs with different numbers of inference requests and finetuning tokens. Data points with a zero finetuning throughput report the latency when serving only inference requests, while other data points mix inference and finetuning workloads. The green line shows the peak finetuning throughput.

troduces significant computation overheads due to the very large model size. *Parameter-efficient finetuning* (PEFT) methods [20] are proposed to solve this issue. Drawing inspiration from the pronounced sensitivity of LLMs to prompts as highlighted in [54], a series of studies introduce trainable prompt embeddings prepended to the input text or attention components while preserving the original LLM parameters [34, 36, 40]. [52] and [28] propose *adapter* modules to introduce new task-specific parameters, which are inserted into the Transformer layers inside the LLM. LoRA [29] leverages the low-rank decomposition concept to construct trainable parameters inserted into the original LLM weights. $(IA)^3$ [38] proposes to scale the pre-trained model weights of an LLM with a trainable vector. Of late, there has been a surge in the proposal of unified approaches that amalgamate various PEFT methods by leveraging human heuristics [26] or employing neural architecture search [42, 76, 78]. Existing PEFT approaches focus on optimizing model performance while minimizing trainable parameters. However, reducing the number of trainable parameters does not inherently imply a corresponding reduction in memory footprint.

2.2 PEFT-based LLM Inference

LLM inference is increasingly essential in both research and industrial applications. Unfortunately, deploying high-quality LLMs without exhausting the available GPU resources can be difficult due to high computational demands and memory requirements. One of the key challenges in existing LLM inference systems is the *memory-intensive* auto-regressive decoding mechanism, since it requires accessing all of the model's parameters to generate each token. Several techniques have been proposed to overcome the memory-bandwidth limitation, such as quantization [66], parallelization [70], batching [58], and kernel optimizations [16]. However, the memory-bandwidth problem still exists, especially in scenarios where both latency and response quality matter. At inference time, PEFT, which unlocks a resource-efficient way to finetune LLMs for specific tasks, also suffers from the same memory access challenges as backbone LLMs.

Recent works have introduced a diversity of systems for simultaneously serving multiple PEFT-based LLMs. PetS [77] is the first unified framework for multi-task Parameter-Efficient Transformers (PETs) serving, which decouples linear layers in the computation graph into task-agnostic shared operations and task-specific PET operations, then uses a high-performance inference engine an innovative operator scheduling strategy to improve concurrency and throughput. While PetS only supports encoder-only models, recent work also focuses on optimizations for generative inference models. PUNICA [10] is a system to serve multi-tenant LoRA in a shared GPU cluster. It proposes a new CUDA kernel called Segmented Gather Matrix-Vector Multiplication, which allows batching GPU operations for the concurrent execution of multiple, different LoRA models. S-LORA [57] is optimized on the kernel introduced by PUNICA. It proposes unified paging, heterogeneous batching, and a novel tensor parallel algorithm to serve thousands of LoRA components on a single machine. However, all these systems are also bounded by memory access like conventional LLM serving systems [32, 70].

3 Co-serving LLM Inference and PEFT

Previous studies have proposed different methods for scheduling heterogeneous workloads (in our case, inference and PEFT) on shared GPU resources. These methods can be classified into three categories: *resource isolation*, *temporal sharing*, and *spatial sharing*. This section will discuss the limitations of these approaches when it comes to serving LLM inference and PEFT, and highlight the critical differences between FlexLLM's co-serving solution and previous work. Figure 3 illustrates the GPU execution timeline of different scheduling approaches for serving inference and finetuning.

Resource isolation. A straightforward approach to support both inference and finetuning is *resource isolation*. This involves partitioning the available GPUs into two isolated groups, each handling one type of request (Figure 3(a)). Resource isolation guarantees that throughput-sensitive finetuning requests do not interfere with latency-sensitive inference requests. When finetuning an LLM, users usually provide a dataset of requests to train the PEFT layers. All the requests in the dataset are submitted to FlexLLM at once, making it easier to build larger batches that can be consumed sequentially. On the other hand, each user submits inference requests independently and expects them to be processed within a predefined SLO, and the arrival time of each request is unknown. Unfortunately, resource isolation can introduce significant GPU under-utilization for workloads with fluctuating request arrival rates, because finetuning and inference requests cannot be batched together when the number of pending requests of a

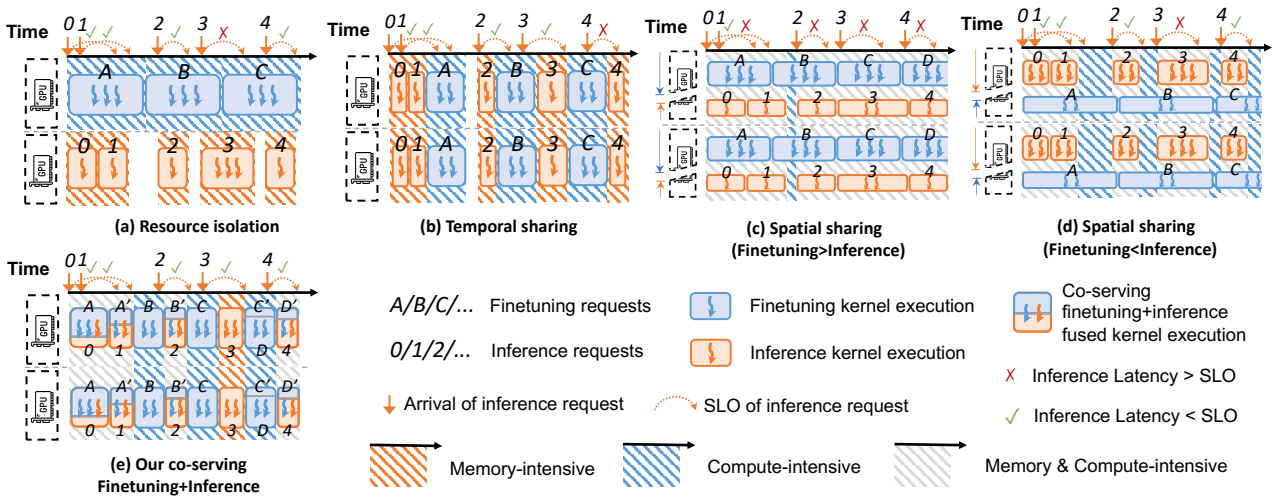

Figure 3: Comparison of different sharing approaches for serving PEFT and inference simultaneously. For spatial sharing and our co-serving approaches, the height of rounded rectangles illustrates the splitting ratio of GPU resources (e.g., SMs or threads).

given type is insufficient to fill a batch on its own. With fewer GPUs available to parallelize the inference computations, it may not be possible to serve requests fast enough to meet their SLO (e.g., request r_3 in Figure 3(a)).

Temporal sharing. Compared to resource isolation, a *temporal sharing* [67, 68] approach can increase GPU utilization by allowing inference and finetuning tasks to share GPU resources through time slices of different lengths depending on various parameters. This technique allows inference and finetuning requests to access all GPUs in the cluster in turns, as shown in Figure 3(b). Using all GPUs to parallelize computation (e.g., with data parallelism or tensor model parallelism) can help reduce the latency of each request, which in turn can help to meet inference SLO restrictions (e.g., inference request r_3 cannot be completed in Figure 3(a), but can be completed in Figure 3(b) by leveraging more resources). However, the arrangement of time slices can also be harmful because of the unpredictable arrival times of inference requests, which may have a stringent latency requirement that cannot be met (e.g., r_4 has been delayed in Figure 3(b) by pre-scheduled finetuning requests).

Spatial sharing. An issue affecting both resource isolation and temporal sharing is that they do not take into account the different characteristics of *compute-intensive* finetuning tasks and *memory-intensive* inference tasks. This results in a significant waste of resources for both workloads. An alternative solution is *spatial sharing*, which allows running different tasks simultaneously, with each task occupying a fraction of GPU resources (i.e., streaming processors). For example, spatial sharing can be done on NVIDIA GPUs by writing multi-stream code or taking advantage of the Multi-Process Service (MPS) [4]. Further, on the latest Hopper and Ampere architectures, spatial sharing can take advantage of multi-instance GPU (MIG) [3] to adjust the partition configuration dynamically. However, the adjustment is not flexible or efficient [2, 19, 69] enough to accommodate the dynamic inference workload. Figures 3(c) and (d) show two separate cases where a larger fraction of GPU resources is assigned to finetuning or inference. While over-provisioning for inference tasks may seem to provide better SLO guarantees, it still fails in the event of unexpected and urgent requests like r_3. Additionally, over-provisioning wastes memory bandwidth resources and significantly slows down the finetuning process. Therefore, spatial sharing cannot be relied upon to run finetuning requests while also serving inference requests from online workloads with satisfactory performance [12, 75].

FlexLLM's co-serving approach. As shown in Figure 3(e), FlexLLM takes a co-serving approach using a fine-grained scheduling mechanism to combine inference and finetuning requests adaptively. In each co-serving iteration, FlexLLM jointly serves inference and finetuning requests by (1) computing keys and values for inference requests in the pre-filling phase, (2) generating a new token for inference requests in the decoding phase, and (3) performing a forward and a backward pass for a subset of tokens from each finetuning request. FlexLLM fuses inference kernels with finetuning forward kernels to minimize kernel launch overhead and reduce accesses to GPU device memory to retrieve model weights. In each iteration, FlexLLM dynamically adjusts the allocation of inference and finetuning tokens, strategically balancing the need to meet the SLOs of inference requests with the goal of maximizing GPU utilization. This adaptive approach ensures that Collie adheres to the requirements of inference requests while optimizing the use of available GPU resources.

Compared to existing approaches, co-serving has two key advantages. First, FlexLLM achieves much higher GPU utilization than prior work. Both resource isolation and temporal

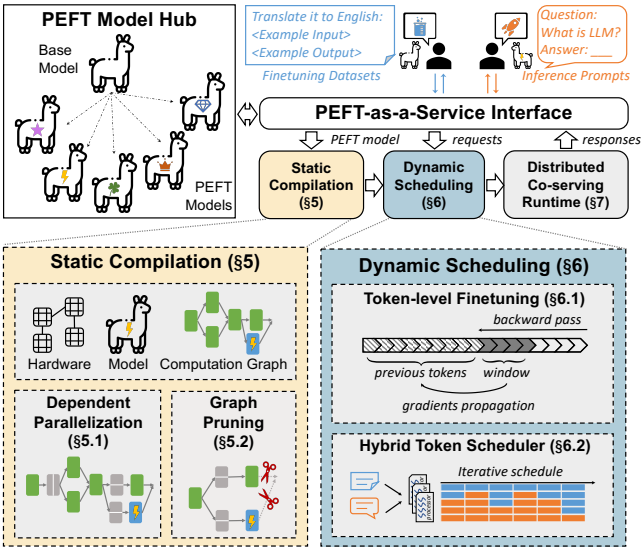

Figure 4: An overview of FlexLLM.

sharing restrict each GPU to handle a single task type at any given time, leading to suboptimal utilization. This inefficiency stems from the fact that inference and finetuning tasks are constrained by GPUs' compute and memory capabilities. Current spatial sharing mechanisms *statically* divide GPU resources for inference and finetuning, which cannot efficiently utilize GPUs for inference due to their inherently unpredictable nature. On the other hand, co-serving can be considered as a *dynamic*, *iteration-level* spatial sharing approach that adaptively adjusts resource allocation between inference and finetuning tasks in accordance with the inference and finetuning workloads.

Second, co-serving maximally retains the SLOs of inference tasks compared to temporal and spatial sharing. Temporal sharing violates stringent inference latency requirements when finetuning requests are pre-scheduled, and spatial sharing only allocates a subset of GPU resources for inference and therefore cannot effectively handle bursty inference requests. Co-serving addresses these issues by adaptively adjusting resource allocation based on the SLOs and the current workload of inference tasks.

4 System Overview

Figure 4 shows an overview of FlexLLM, which co-serves PEFT and inference requests for LLMs. Users can choose to perform either finetuning or inference services on a model from the PEFT *model hub*, which stores the backbone LLM and all finetuned models. The programming interface for both inference and finetuning requests is unified through a *PEFT-as-a-Service (PEFTaaS) interface*. FlexLLM comprises three main system components. First, for finetuning requests that execute in parallel across multiple GPUs, the static compilation module (§5) generates a parallel computation graph. This graph specifies the execution of the PEFT model over the distributed environment and optimizes the graph by pruning unnecessary tensors for memory saving. Second, the dynamic scheduling module (§6) adapts a token-level finetuning mechanism. It mixes inference and finetuning tokens with a hybrid scheduling policy. Finally, the computation graph and schedule plan are executed by FlexLLM's distributed co-serving runtime (§7).

4.1 PEFT-as-a-Service Interface

To facilitate the sharing of the backbone LLM between inference and finetuning, FlexLLM represents a PEFT model as a sequence of *bypass networks* attached to the backbone LLM. Each bypass network takes a *single* tensor from the backbone LLM as input and produces a *single* output tensor, which is added to one tensor of the backbone LLM. Let X and Y denote the input and output tensor of the bypass network, and let $f_B(X)$ and $f_A(X)$ denote the neural architecture of the backbone and bypass network for calculating Y. The bypass network can be formulated as $Y = f_B(X) + f_A(X)$. All existing PEFT methods can be represented in this format. For instance, the $(IA)^3$ architecture modifies the topology of the backbone LLM by inserting an elementwise multiplication operator. It can be transformed into FlexLLM bypass network format since $Y = X \odot W = X + X \odot (W - O)$, where $\odot$ is elementwise multiplication and O is a matrix of the same shape as W whose elements are all one. A significant benefit of this interface is that all PEFT models preserve the neural architecture of the backbone LLM, which enables FlexLLM to fuse the computation graphs of different PEFT models. FlexLLM provides a unified user interface to launch inference and finetuning tasks by specifying a PEFT model and committing different types of input data, such as finetuning datasets or inference prompts.

5 Static Compilation

After a PEFT model is registered, FlexLLM compiles it into a parallel computation graph (PCG) [64]. To optimize the model's finetuning across multiple devices, the system generates the best parallelization strategy by applying graph transformations to the PCG. Two key static compilation optimizations are *dependent parallelization* (Section 5.1) and *graph pruning* (Section 5.2). Dependent parallelization allows all PEFT models from the same backbone LLM architecture to share the backbone model parameters. The system generates the best parallelization plan for each PEFT based on the parallelization of the backbone LLM model. Graph pruning is another optimization technique that reduces the number of intermediate activation tensors that need to be saved during the forward pass and kept until the backward pass has completed. This technique helps to further reduce the memory requirements of finetuning the PEFT models. The rest of this section elaborates on the two techniques in depth.

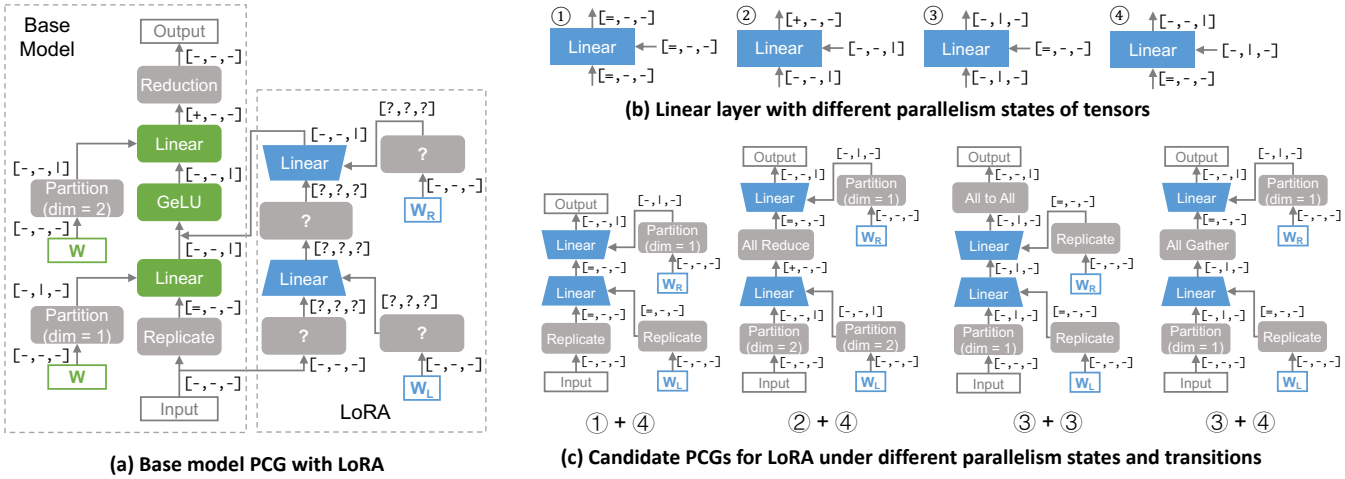

Figure 5: Illustration of FlexLLM's different dependent parallelization strategies with the example of LoRA.

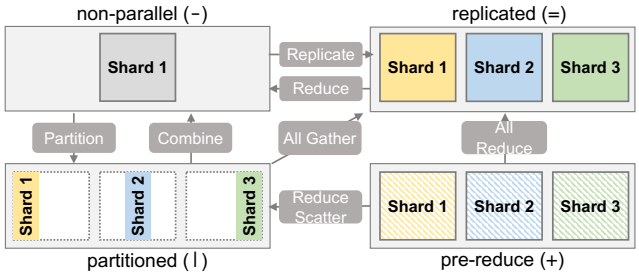

Figure 6: Four possible parallel states for a tensor dimension and their transitions. For each parallel state, the symbol in parenthesis shows the notation FlexLLM used to represent it.

5.1 Dependent Parallelization

不同 inputs 会得到不同 activations，怎么 share?

To minimize memory overhead, FlexLLM enforces that all finetuning and inference tasks share the weights and intermediate activations of the base LLM, which requires the parallelization strategies of the PEFT models to be compatible with that of the base LLM, a task we called *dependent parallelization*.

To represent a parallelization strategy, FlexLLM generalizes the *parallel computation graph* representation introduced in Unity [64]. Similar to existing distributed LLM systems, FlexLLM supports parallelizing the base LLM using *tensor-model parallelism*, as introduced in Megatron-LM [59]. We first present Unity's PCG representation and next describe how FlexLLM extends this representation. We also refer readers to [64, §3] for a more comprehensive description of PCGs.

Parallel computation graphs (PCGs). PCGs leverage four *parallelization primitives* (i.e., `partition`, `combine`, `replicate`, and `reduce`) to collectively capture the communication pattern, data movement, and computation distribution associated with different parallelization strategies. Each tensor in a PCG is represented by a set of data dimensions, each of which includes two fields: a size and a degree. The degree field indicates the number of partitions the tensor is divided into along that dimension. Each tensor also includes a leading *replica* dimension, specifying the number of replicas of the tensor. A replica degree larger than 1 indicates that the tensor is replicated across multiple devices.

PCGs use four parallelization primitives to represent the computation and communication costs associated with different parallelization strategies. First, `partition` and `combine` change the parallelism degree of a tensor and are the "back propagation" of each other. In particular, `partition` increases the parallelism degree of a tensor dimension by equally splitting the dimension into multiple partitions, while `combine` reduces a tensor's parallelism degree by concatenating multiple partitions into one. Second, `replicate` and `reduce` update the number of replicas of a tensor by copying and summing it. A more comprehensive introduction to the PCG primitives is available in the Unity paper [64, Figure 8].

Unity generates PCGs by iteratively applying graph substitutions on an input, unparalleled computation graph, which allows Unity to maintain the internal states of a tensor's data dimensions implicit through the sequence of applied graphs substitutions. However, Unity's approach requires starting from a fully unparalleled computation graph and does not work for dependent parallelization, where the base model has already been parallelized based on user-provided specification.

base model 的并行由用户指定

To address this issue, FlexLLM extends the PCG representation by introducing a *state* field for each dimension of a tensor. Each tensor dimension can be in one of the following four states, as shown in Figure 6. First, the *non-parallel* state indicates the tensor is not parallelized along this dimension (i.e., a parallelism degree of 1). Second, the *partitioned* and *replicated* states indicate the tensor is partitioned along the given dimension by splitting and replicating the original

tensor. Finally, the *pre-reduce* state refers to the case where different replicas of the tensor each hold partial results, and a reduction across these replicas recovers the tensor's final output. This scenario is common when parallelizing LLMs using tensor model parallelism (see an example for parallelizing the base model in Figure 5(a)).

Dependent parallelization problem. Given multiple input and output parallel tensors and the parallelism states of their data dimensions, FlexLLM discovers a PCG that produces the output tensors with the given parallelism degrees and states by taking the input tensors as input.

Unlike the parallelization optimization problems addressed by existing auto-parallelization approaches, FlexLLM solves an optimization problem with additional constraints. Figure 5(a) shows the PCG for parallelizing the two linear layers using tensor model parallelism. Each green (or gray) box indicates a tensor algebra (or parallelization) operator, and each edge between operators represents a parallel tensor, whose dimensions are shown next to the parallel tensor. In this example, all parallel tensors have three dimensions, the first of which indicates the replica dimension and the remaining two correspond to the rows and columns of a matrix. We consider the dependent parallelization task for applying LoRA to the first linear operator, whose input and output tensors are parallelized in the `[-, -, -]` and `[-,-,|]` states, respectively. For the output tensor, the final partitioned state indicates the column of the output matrix is split across GPUs.

目标是根据 base model 的并行决定 adapter 的并行

FlexLLM leverages the parallelism states of the input and output tensors for generating PCGs for the LoRA network. Specifically, FlexLLM inserts parallelization operators between tensor algebra operators and leverages the parallelism states of each tensor's data dimensions to type-check the compatibility of these parallelization operators. Figure 6 shows how parallelization operators transit a tensor's parallelism states. Most PEFT models only contain elementwise operators, which do not change the parallelism states of a tensor, and linear operators, which update an input tensor's parallelism states by following the rules given by Figure 5(b).

To generate PCGs for a bypass network, FlexLLM inserts a parallelization operator between all tensor algebra operators, enumerates the type of the parallelization operator, and examines the compatibility of all tensors in the new PCG. A PCG that passes this examination is considered a *candidate PCG* for the dependent parallelization task. Figure 5(c) shows four candidates PCGs discovered by FlexLLM for parallelizing a LoRA network. These candidate PCGs introduce different parallelization operators and communication overheads. FlexLLM reuses the profiling-based cost model of Unity [64] and selects the candidate PCG with the lowest cost. For the four candidate PCGs in Figure 5(c), the first one achieves the lowest cost, since it does not introduce communications during forward or backward. Note that the `replicate` operator after the input tensor does not introduce cost since it can be fused with that of the base model. The other two parallelization operators introduce no communication cost since they are independent from the input and can be pre-processed.

Algorithm 1 Static graph pruning. For an operator n, UPDATEINPUT$(n, O(n))$ returns a set of input tensors needed in order only to compute $O(n)$ of the operator.

Inputs: PCG of a PEFT model $\mathcal{G}$
Outputs: $\mathcal{A}$ is a set of tensors to be memorized, $\mathcal{R}$ is a set of tensors to be rematerialized
▷ Step 1: computation graph pruning
$\overline{\mathcal{G}} =$ REVERSEAUTODIFF$(\mathcal{G})$
$\mathcal{Q} = \emptyset$ ▷ $\mathcal{Q}$ is a queue of updated operators
for operator $n \in \overline{\mathcal{G}}$ **do**
  for output tensor $t \in O(n)$ **do**
    if t is the weight gradient of the base LLM **then**
      $O(n) = O(n) \setminus \{t\}$
      $I(n) =$ UPDATEINPUT$(n, O(n))$
      $\mathcal{Q}$.push_back(n)
while $\mathcal{Q}$ is not empty **do**
  $n = \mathcal{Q}$.pop_front()
  for output tensor $t \in O(n)$ **do**
    if $\nexists u.t \in I(u)$ **then**
      $O(n) = O(n) \setminus \{t\}$
      $I(n) =$ UPDATEINPUT$(n, O(n))$
      $\mathcal{Q}$.push_back(n)
$\mathcal{A} = \emptyset$
for operator $n \in \mathcal{G}$ **do**
  for tensor t *in* $O(n)$ **do**
    if $\exists u \in \overline{\mathcal{G}}.t \in I(u)$ **then**
      $\mathcal{A} = \mathcal{A} \cup \{t\}$
▷ Step 2: opportunistically rematerializing tensors
for tensor $t \in \mathcal{A}$ **do**
  Let n be the operator that outputs t (i.e., $t \in O(n)$)
  if $I(n) \subseteq \mathcal{A}$ and COST$(n) <$ *threshold* **then**
    $\mathcal{A} = \mathcal{A} \setminus \{t\}$, $\mathcal{R} = \mathcal{R} \cup \{t\}$
return $\mathcal{A}, \mathcal{R}$

5.2 Graph Pruning

This section introduces FlexLLM's *graph pruning* algorithm, which takes a PEFT model and its backbone LLM as input and returns a minimal set of intermediate activations that must be reserved to finetune the given bypass networks. Graph pruning requires reasoning about the input and output tensors of each operator to prune nodes and edges, therefore FlexLLM introduces the following notations to the PCG representation. Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ denote the PCG of a PEFT model, where each node $n \in \mathcal{N}$ is a tensor algebra (or parallelization) operator and each edge $e = (n_1, n_2) \in \mathcal{E}$ is a tensor shared between operators. For each node n, let $I(n)$ and $O(n)$ denote its set of input and output tensors. Therefore, $(n_1, n_2) \in \mathcal{E}$ if and only if $O(n_1) \cap I(n_2) \neq \emptyset$ (i.e., n_2 depends on n_1).

剪枝不需要的中间激励

The design of FlexLLM's graph pruning is based on two

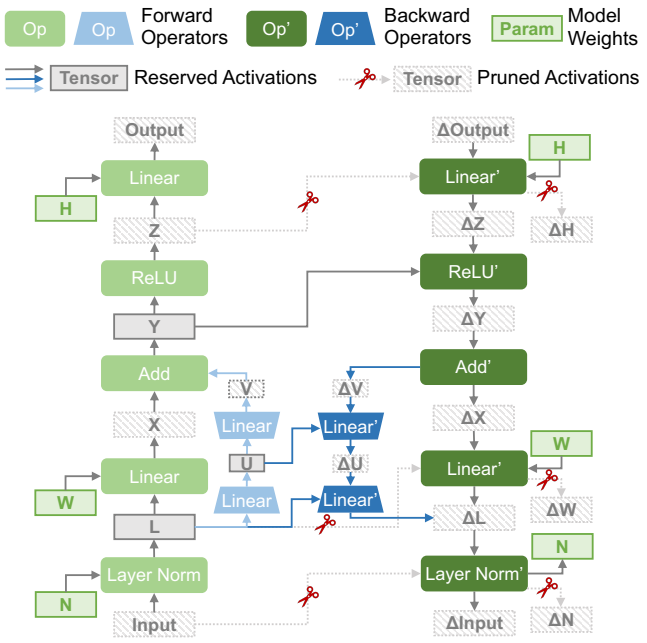

Figure 7: Static graph pruning for an MLP model with LoRA.

Adapter params 不需要 base layer 的 activation 就可以计算自己的梯度

不同 job 的 adapter 绑定的位置不同，需要的激励也不同

important observations. First, conventional ML training procedures maintain all intermediate activations in a forward pass for calculating gradients during the subsequent backward pass. However, due to the linear algebra nature of most operators, *intermediate activations* are reserved for computing the gradients for the *trainable parameters* of the base LLM, which are frozen during PEFT. This property allows FlexLLM to compute the gradients of the bypass networks of a PEFT model *without* using the majority of the intermediate activations. Second, different PEFT models attach bypass networks to different positions of the backbone LLMs and therefore require different intermediate activations for backpropagating through the bypass networks (see examples in Figure 8).

Based on these observations, FlexLLM performs *static* graph pruning when generating the PCG for each registered PEFT model and *dynamically* schedules the finetuning and inference tasks of these PEFT models and allocates memory (see Section 6). Algorithm 1 shows FlexLLM's graph pruning algorithm, which takes the PCG of a PEFT model $\mathcal{G}$ as input and generates an execution plan that specifies how to cache or recompute the intermediate activations during a forward pass for backpropagation. First, REVERSEAUTODIFF constructs the backward graph $\overline{\mathcal{G}}$ using the reverse-mode automatic differentiation algorithm [27]. Next, since all weights of the base LLM are frozen during finetuning, FlexLLM prunes gradients for these weights. Pruning an operator's output tensors can enable subsequent pruning opportunities. FlexLLM maintains a queue Q to keep track of all operators whose input and/or output tensors have been updated and iteratively examines these operators to discover additional pruning opportunities. After graph pruning, FlexLLM generates a set of tensors $\mathcal{A}$ that contains all activations that need to be maintained for backward propagation. FlexLLM further combines graph pruning with two established memory optimizations: *rematerialization* and *activation compression*.

Rematerialization. Tensor rematerialization [30] consists of selectively discarding some tensors in the forward pass and recomputing them during backward propagation. Originally designed to reduce the peak memory requirement of DNN training, this technique is also beneficial for LLM co-serving. For each remaining tensor $t \in \mathcal{A}$ after graph pruning, FlexLLM rematerializes t if and only if (1) all input tensors of t are stored in $\mathcal{A}$ and (2) recomputing t using the input tensors introduces a low rematerialization overhead.

Activation compression. In addition, FlexLLM also opportunistically applies *lossless* compression for remaining tensors whenever possible. This is because performing backward computation for some operators does not require access to the original input tensors. One example is the `ReLU` operator

$$y = \texttt{ReLU}(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

一些 op 后向计算时不需要访问原来的 input tensors

whose derivative is

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Therefore, instead of storing the original input tensor x, FlexLLM keeps the bitmask of x. Note that this compression does not introduce any accuracy loss.

6 Dynamic Scheduling

After static compilation generates a PEFT model's executable generalized PCG, FlexLLM needs to determine the execution schedule to handle finetuning and inference requests concurrently. This is a challenge because of the distinct characteristics of each workload. For one, inference requests are received in a online fashion and are dynamically fluctuating, necessitating a system architecture that can rapidly adjust the GPU resources to ensure low latency. On the other hand, finetuning involves both a forward pass (like inference) and a backward pass to update model parameters. Third, the auto-regressive inference for generative LLMs requires a memory-intensive token-by-token generation pattern, while finetuning typically employs a compute-intensive sequence-level approach for better throughput. To address these challenges, FlexLLM employs a novel *token-level finetuning* mechanism and a *hybrid token scheduler*.

6.1 Token-Level Finetuning

To maximize GPU utilization, existing LLM finetuning systems generally process finetuning samples at the *sequence*

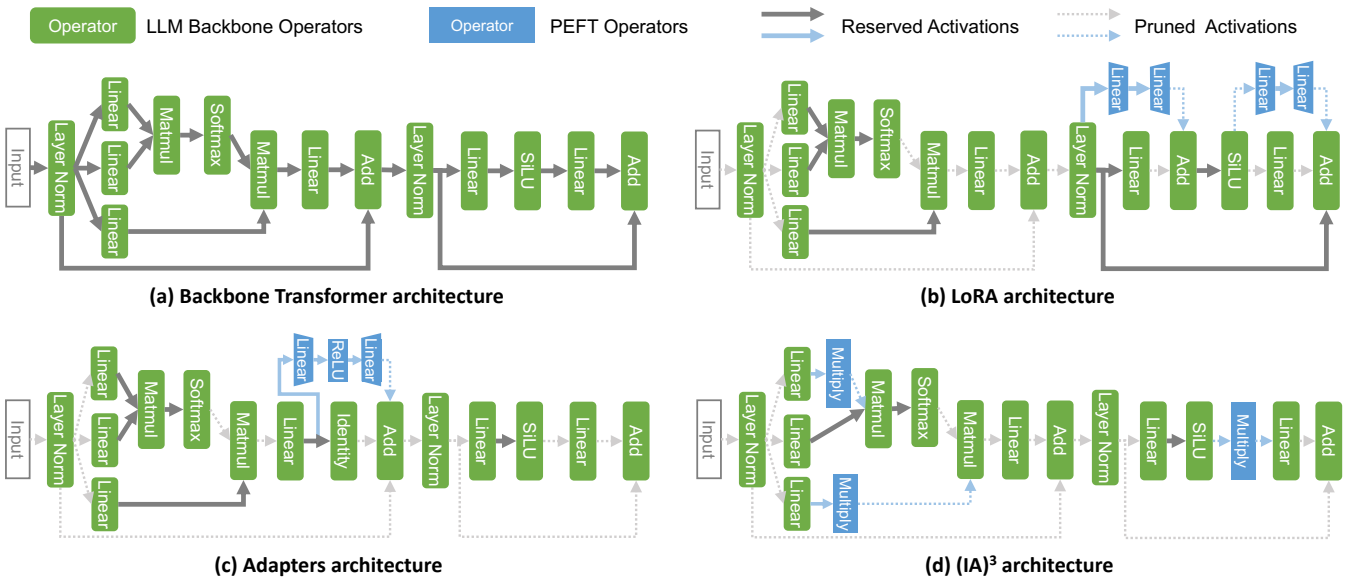

Figure 8: An overview of existing parameter-efficient fine-tuning (PEFT) methods. Green boxes show the operators of the backbone LLM, while blue boxes are the operators introduced by different PEFT methods. Arrows represent intermediate activations to be reserved and the dashed arrows demonstrate they are pruned by FlexLLM.

level rather than the *token* level. As a result, directly batching inference requests with finetuning requests significantly increases the latency of each iteration and affects the SLO of inference requests.

We present our *token-level* finetuning mechanism in Algorithm 2. The main idea is to divide the finetuning process into smaller steps using a dynamic sliding window that consists of a series of tokens. This window is used for both the forward pass and the backward pass, and its size is determined by the hybrid token scheduler (line 4 and line 13), to ensure that the co-served inference tokens do not exceed the SLO constraints (§6.2). During the forward pass (lines 3-11), in each step, we slice the finetuning request into windows of s_i tokens (line 5) and feed them into the model. The model processes the tokens layer by layer (lines 7-8), computes the generative loss of these tokens (line 10), and then moves to the next window of tokens (line 11). To maintain the same compute semantic as the sequence-level finetuning, we need to cache keys and values (similar to incremental decoding for inference [70]) as well as queries (reused by backward attention calculation) following the *causal mask* of autoregressive LLMs.

每个 step 只做一部分 tokens 的 finetune

In the backward pass (lines 12-20), we get a slice of s_j tokens' loss in each step (line 14) and execute the model layers in reverse order to calculate the gradients (lines 15-16). However, unlike sequence-level attention execution, our token-level backward attention execution is more complex because of the auto-regressive token dependencies. We illustrate both the forward and backward PCGs of the attention module on the left and right sides of Figure 9. When we get a slice of s_j tokens' loss, the generated gradients of queries, keys, and values (i.e., ΔQ, ΔK, ΔV) should have a shape of $[s_j, h]$, where h is the hidden size. The ΔQ calculation requires access to the key cache of all previous l_j tokens, which aligns with the causal mask in forward execution. However, the autoregressive execution also makes the generated ΔV and ΔK be the size of $[l_j, h]$ because the tokens' loss within the window also contributes to all previous tokens. As shown in Figure 10, we propose to accumulate the KV gradients in each step and only apply the fully accumulated s_j gradients (i.e., size of $[s_j, h]$) to calculate the other model gradients until the backward traversal of all tokens in the sequence is completed.

Figure9 没看太懂

6.2 Hybrid Token Scheduler

等到一个 seq 所有 tokens 的 bp 做完再累积 update

FlexLLM's *hybrid token scheduler* schedules the token-level GPU computations in two steps. First, FlexLLM decides the schedule of available inference requests based on specific scheduling policies. FlexLLM uses Orca's *iteration-level scheduling* [70] as the default scheduling policy, which maintains a fixed maximum batch size b and dynamically replaces each completed request with a new one whenever available. Other scheduling policies, such as assigning different priorities to requests of different lengths, or partitioning the prefilling phase of long input prompts into multiple pieces, are also feasible for FlexLLM. Second, after determining the inference schedule in each iteration, FlexLLM appends to the batch as many finetuning tokens as possible (i.e., sliding window size of s) for maximum resource utilization. The number of finetuning tokens that can be added is determined automatically using the formula $s = \arg\max f(c,s) \leq \text{SLO}$, where $f(\cdot,\cdot)$ is the latency estimation function and c is the number of

Forward 的输出是 token group 粒度 (本来为 seq 粒度，即 (out_seq_len, hidden)) 的，使用 generative loss 来计算该组的 loss，并对该 token group 进行 backward 计算 gradients。等到该 seq 内所有 tokens 的 gradients 都算完了再一起更新参数。

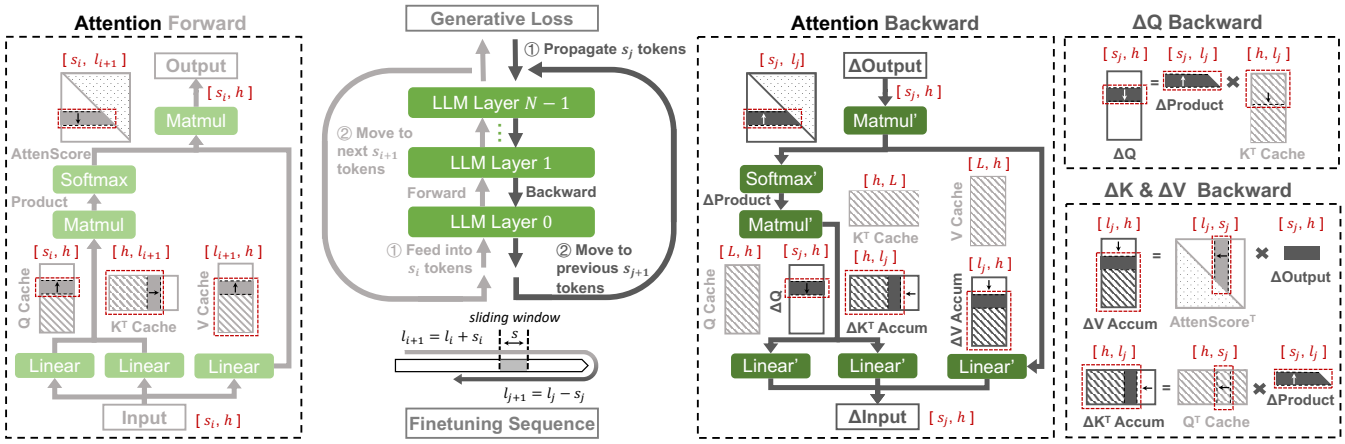

Figure 9: Illustration of attention module's forward and backward execution with FlexLLM's token-level finetuning mechanism.

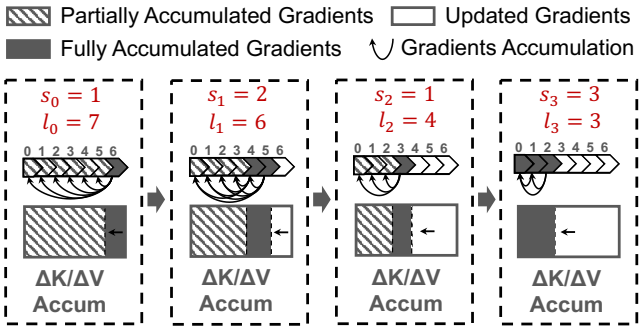

Figure 10: Illustration of KV gradients accumulation in the backward pass of FlexLLM's token-level finetuning mechanism.

inference tokens that are running in the current iteration. The inference latency can be cheaply estimated by profiling the LLM execution costs in advance [46] so that the scheduling overhead is negligible compared to the per iteration latency.

Figure 11 illustrates the execution timeline of FlexLLM and shows an example of a token scheduling plan within a finetuning mini-batch r_A under a particular inference requests arrival pattern (i.e., r_0 to r_9). To maintain the same finetuning semantic, FlexLLM's scheduling follows the execution dependency between each mini-batch's forward and backward pass. Within the forward pass, FlexLLM uses fused kernels to process both inference and finetuning tokens, avoiding additional kernel launch overhead. This is possible since the requests follow the same token-wise compute logic. Regarding the attention operator, finetuning requests follow the same logic and causal masking as inference requests in the pre-filling phase. For the backward pass, FlexLLM adapts a space-sharing approach, using two separate streams to handle inference and finetuning tokens.

Algorithm 2 Token-level finetuning mechanism.

Inputs: PEFT model M depth of N; Fine-tuning request r with sequence length of L
Outputs: PEFT model gradients ΔM
for $i \leftarrow 0, l_0 \leftarrow 0$; $l_i < L$; i++ **do** ▷ Forward pass
  $s_i \leftarrow \text{HybridTokenScheduler}(l_i, L)$
  $r_i \leftarrow \text{Slice}(r, l_i, s_i)$
  $X_{0,i} \leftarrow \text{Tokenize}(r_i)$
  for model layer index $n \in \text{Range}(0, N)$ **do**
    $X_{n+1,i}, Q_{n,i}, K_{n,i}, V_{n,i} \leftarrow \text{Forward}(M_n, X_{n,i})$
    $QKVCache_n \leftarrow \text{Append}(Q_{n,i}, K_{n,i}, V_{n,i})$
  $Loss_i \leftarrow \text{GenerativeLoss}(X_{N,i}, r_i)$
  $l_{i+1} \leftarrow l_i + s_i$
for $j \leftarrow 0, l_0 \leftarrow L$; $l_j > 0$; j++ **do** ▷ Backward pass
  $s_j \leftarrow \text{HybridTokenScheduler}(l_j, 0)$
  $Y_{L-1,j} \leftarrow \text{Slice}(Loss, l_j, s_j)$
  for model layer index $n \in \text{Range}(N, 0)$ **do**
    $Y_{n,j}, \Delta K_{n,j}, \Delta V_{n,j} \leftarrow \text{Backward}(M_n, Y_{n-1,j}, QKVCache_n, \Delta KVAccum_n)$
    $\Delta KVAccum_n \leftarrow \text{Add}(\Delta K_{n,j}, \Delta V_{n,j})$
    $G_{n,j} \leftarrow \text{Slice}(\Delta KVAccum_n, l_j, s_j)$
    $\Delta M_n \leftarrow \Delta M_n + \text{CalculateGrads}(M_n, G_{n,j})$
  $l_{j+1} \leftarrow l_j - s_j$
return ΔM

7 Implementation

FlexLLM is built on top of FlexFlow Serve [44] with 9K lines of C++ and CUDA code. It comes with both C++ and Python bindings, and its Python interface is fully compatible with HuggingFace's `PEFT` [41] and `transformers` [65] libraries. This means that users can easily serve models imported directly from HuggingFace, and even upload finetuned PEFTs to the HuggingFace Hub. Moreover, FlexLLM supports any PEFT models with LLAMA [62], GPT [9], OPT [74], Fal-

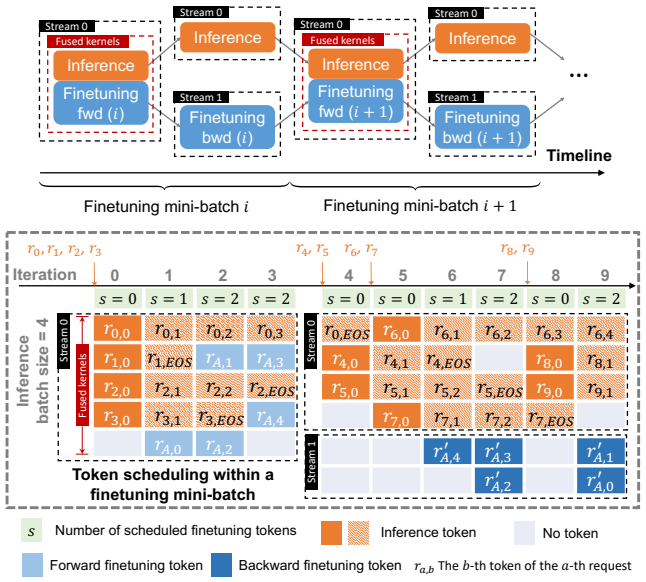

Figure 11: Execution timeline and an example of token scheduling when co-serving inference requests and a finetuning mini-batch.

con [7], or MPT [61] backbones. In addition, we provide instructions on how to add support for custom models from HuggingFace. All the code was integrated into FlexFlow, available at https://github.com/flexflow/FlexFlow/. Further, FlexLLM is available to the users as a part of the FlexFlow `pip` package.

Memory management. FlexLLM manages the required GPU memory using a combination of static and dynamic allocation. Static allocation reserves GPU space for the backbone weights and the KV cache required for incremental decoding during inference. Dynamic allocation is used for gradients, activations, and optimizer states. When a finetuning request is received, FlexLLM allocates memory during the first forward pass and reuses it during subsequent backward passes, freeing up space when it's no longer needed.

Key-value gradient accumulator. To support token-level finetuning, it is necessary to accumulate key-value gradients for every preceding token in a given sequence during backpropagation. To achieve this, FlexLLM utilizes static allocation to reserve space for key-value gradient accumulation. During backpropagation, the gradients for keys and values of a specific segment are obtained, and the position of this segment in the overall sequence is determined. FlexLLM then accumulates the new key-value gradients of all subsequently scheduled segments. When a segment is scheduled during backpropagation, it includes the accumulated gradients brought by all future tokens.

Operator implementation We have optimized kernels for NVIDIA GPUs using cuBLAS libraries for linear algebra operators and cuDNN for the other supported operators to run the most popular LLMs. Additionally, we have custom kernels for the remaining computations (such as layer normalization and SiLU activations). To reduce kernel launch overhead, we aggressively fuse customized kernels within and across operators. For example, we use a single kernel to compute the attention score for the inference tokens in the generative phase. We also fuse each residual summation with the following LayerNorm/RMSNorm operator and the Sigmoid, SiLU, and element-wise tensor multiplications from the LLaMA transformer layer.

8 Evaluation

LLMs and PEFT models. We assessed the performance of FlexLLM by comparing it with existing approaches using three publicly available LLMs: OPT-13B [74], LLaMA-30B [62], and LLaMA-2-70B [63]. We obtained the pre-trained model parameters for the LLMs directly from their HuggingFace repositories [1]. Since existing LLM serving systems only support base LLMs and LoRA, our evaluation focuses on these two types of models to conduct a fair comparison. Like prior works, we applied LoRA (with a rank of 8) to both linear layers of each multi-layer perceptron block in each LLM. With this setup, we obtained, respectively, 15.6M, 30.5M, and 50.1M trainable parameters for the three LLMs.

Platform. All experiments were performed on the Perlmutter supercomputer [5]. Each compute node is equipped with an AMD EPYC 7763 64-core 128-thread processor, 256 GB DRAM, and four NVIDIA A100-SXM4-40GB GPUs. Nodes are connected with HPE Slingshot 200 Gb/s interconnects. We utilized tensor-model parallelism for serving the base LLMs within a compute node, and pipeline parallelism across nodes for all experiments.

8.1 End-to-end comparison

We first conducted an experiment to compare FlexLLM's co-serving approach with existing approaches that utilize separate clusters to perform inference and finetuning. The baseline system we used employed HuggingFace (HF) PEFT [41] for finetuning and S-LoRA [57] for serving PEFT-based inference requests. In this experiment, we varied the inference workload and measured the finetuning throughput that different systems can achieve. We allocated a total of 4, 8, and 16 A100 GPUs for jointly serving inference and finetuning workloads for OPT-13B, LLaMA-30B, and LLaMA-2-70B, respectively. Each inference request had an average input prompt length of 256 and generated 64 tokens. We used a per-token inference latency of 50ms, 80ms, and 110ms as the SLA for inference requests. Each finetuning request had a sequence length of 1024 and used the Adam optimizer [31]. Figure 12 shows the result. We measured the number of inference tokens generated per second to quantify the inference workload. The baseline approach used a minimum number of GPUs to serve inference requests and allocated all remain-

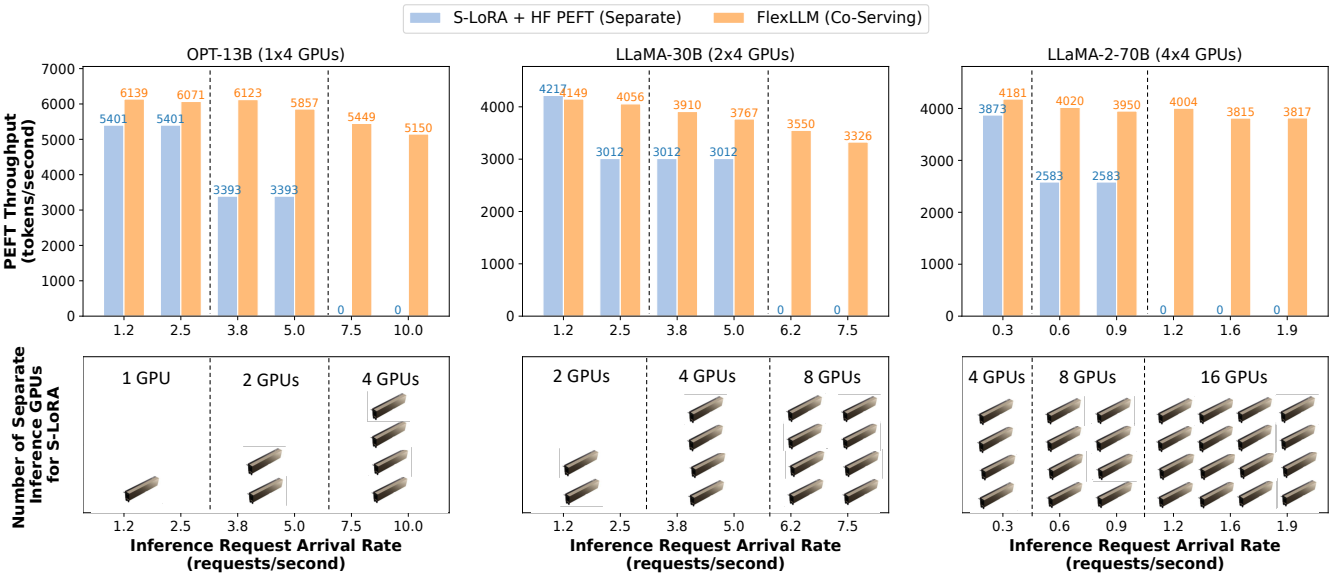

Figure 12: End-to-end comparison between co-serving and using separate resources on three models.

ing GPUs for finetuning PEFT models for a given inference workload.

Under lightweight inference workloads, FlexLLM achieves on-par performance with the baseline approach that uses separate clusters. This is because the baseline approach reserves most of the GPUs for fine-tuning PEFT models, using a very large fine-tuning batch size (e.g., 1024 for fine-tuning OPT-13B) when the inference workload is moderate. As a comparison, FlexLLM co-serves inference and finetuning using all GPUs and restricts the maximum finetuning batch size to satisfy the inference SLO. For example, to maintain a per-iteration co-serving latency of 50ms for OPT-13B, FlexLLM can process at most 384 tokens within an iteration.

Under medium inference workloads, FlexLLM achieves up to 1.6× higher finetuning throughput than the baseline. This performance improvement is achieved by utilizing more GPUs for finetuning and using available GPU cycles to process finetuning tokens more efficiently.

Finally, under heavy inference workloads, the existing approach has to allocate all GPUs for inference to cope with the high request arrival rates. As a result, it cannot make any progress on finetuning requests. In contrast, FlexLLM's finetuning throughput only moderately decreases as we increase the inference workload and never falls below 80% of the peak finetuning throughput. Even when the existing approaches cannot make any progress, FlexLLM manages to perform well.

8.2 Memory Optimization

Figure 13 illustrates the (finetuning-only) memory overhead analysis of FlexLLM across various PEFT and full finetuning methods on LLaMA-2-70B, with a sequence length 1024. We have compared the memory requirements of FlexLLM with HuggingFace PEFT and also included the memory usage for inference-only cases as a point of reference. In full-model finetuning, FlexLLM has the same memory requirement as HuggingFace. However, in the case of PEFT (LoRA, Adapter, and IA3), FlexLLM exhibits an impressive reduction of approximately 32%-36% in memory overhead compared to HuggingFace PEFT. This reduction brings FlexLLM closer to the memory overhead observed during inference, which is the lower boundary for finetuning procedures.

FlexLLM introduces a series of memory optimizations to reduce the memory overhead of PEFT finetuning and allow co-serving finetuning with inference workloads. To understand the effectiveness of FlexLLM memory optimizations, we perform an ablation study that incrementally turns off these optimizations (graph pruning, rematerialization, and token-level finetuning) and measures the memory requirements. Since all optimizations preserve model weights and reduce activation memory requirements, we only consider activation memory overheads in these experiments. Figure 14 shows the results. We analyze the activation memory requirements of FlexLLM for different finetuning methods on LLaMA-2-70B and a sequence length of 1024. FlexLLM saves 85%-87% memory requirements for activations compared to HF PEFT. The major improvement comes from graph pruning. With graph pruning only, FlexLLM achieves 71%-74% activation memory overhead reduction compared to HuggingFace PEFT. Rematerialization and token-level finetuning further reduce memory overhead by 0-8% and 4%-10%, respectively.

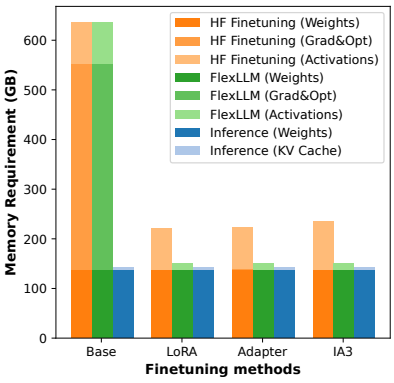

Figure 13: Detailed comparison of finetuning memory overheads.

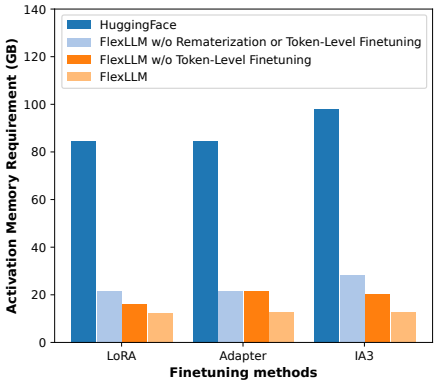

Figure 14: Ablation study of memory saving techniques in FlexLLM.

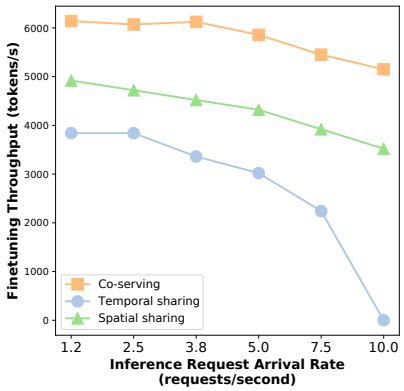

Figure 15: Comparing different scheduling strategies over FlexLLM.

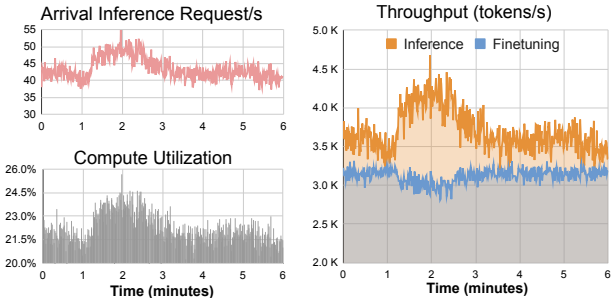

Figure 16: Case study of FlexLLM's system throughput and compute utilization for fluctuating inference workload.

8.3 GPU Scheduling

This section compares FlexLLM's co-serving mechanism with two commonly used GPU scheduling strategies: *temporal sharing* and *spatial sharing*. These two baselines are implemented over FlexLLM but replace the co-serving mechanism with the corresponding approach. To conduct a fair comparison, we have enabled all memory optimizations for all scheduling strategies, so the difference in performance only comes from GPU scheduling. Temporal sharing schedules inference and finetuning computation in an interleaved fashion, as shown in Figure 3. Spatial sharing, conversely, uses separate CUDA streams to launch inference and finetuning computation in parallel. Figure 15 shows that even under a very high inference request arrival rate (i.e., 10 requests per second), the finetuning throughput of our co-serving approach only drops by around 10%, which is still far better than the temporal and spatial sharing strategies.

8.4 Case Study

This section evaluates FlexLLM using a dynamic inference workload generated from a real online production trace at Microsoft Azure [55]. To ensure compatibility with our experiment environments, we replayed a 6-minute length trace segment and re-scaled its arrival intensity, as previous works have done [6, 24, 45, 73]. In the experiment, each inference request consisted of an initial prompt length of 64 tokens, generating 32 new tokens during the inference phase. The inference SLO required that the per-iteration latency did not exceed 50ms. We used a mini-batch size of 1024 tokens for the PEFT finetuning request. We conducted our evaluation using the LLaMA-30B model over 4 A100 GPUs.

In our case study, as shown in Figure 16, we observed that the arrival rate of the inference requests initially increased to a peak level after around two minutes, and then gradually decreased. FlexLLM could automatically detect the fluctuations in the workload and improve the ratio of inference tokens (vs fine-tuning tokens) in each iteration's batch. This significantly increased inference throughput (from 3.5K to 4.7K) and GPU compute utilization (from 21.5% to 25%). The impact on the finetuning throughput was only minimal (less than 0.3K), which further highlights the potential waste of resources in a system that is solely focused on inference.

9 Related Work

ML serving and inference systems. Numerous ML serving systems [15, 24, 25, 33, 51] have been proposed in recent yearsto address various challenges such as latency prediction, system scalability, model performance, swapping [8], preemption [13], workload estimation [73], cost-effectiveness [71], and more. These systems include Nexus [56], which focuses on batching and scheduling multiple small DNN models within a GPU for higher execution throughput, and Gpulet [12], which improves resource assignment and utilization by creating multiple virtual GPUs within a single physical GPU. However, these systems are mainly designed for small CNN models. AlpaServe [37] extends the temporal sharing approach to large models via model parallelism across GPUs. Some concurrent PEFT serving systems like Punica [10] and S-LoRA [57] follow the design of PetS [77] and further improve it by leveraging new GPU kernels for

latest PEFT methods like LoRA [29]. However, most of these existing works ignore the resource utilization issue in LLM inference systems. To the best of our knowledge, FlexLLM is the first PEFT co-serving system that concurrently executes finetuning and inference tasks and improves resource utilization without affecting inference latency.

GPU resource sharing for ML. GPU resource sharing is an emerging field of study focusing on the efficient allocation and management of GPU resources among multiple ML users or applications. Lyra [35] allocates dedicated resources for training and serving with dynamic adjustment. Gandiva [67] and Antman [68] take a time-slicing approach to switch between different training jobs for better cluster utilization. GSLICE [19] targets the over- and under-provisioning problems in multi-steam and MPS approaches by developing an adaptive GPU resource allocation and batching scheme for tasks with different SLO constraints. Although these approaches can not directly solve the resource utilization problem in PEFT service, we believe GPU resource sharing will be increasingly important as GPUs become more powerful.

10 Conclusion

This paper proposes FlexLLM, a system for co-serving parameter-efficient finetuning and inference of large language models. We observe the distinct GPU resource utilization features between finetuning and inference workloads and propose a holistic PEFT serving system that supports to co-serve finetuning requests without affecting inference latency.

References

[1] Huggingface Models. https://huggingface.co/models, 2023. 11

[2] NVIDIA MIG Partitioning Limitations and Resetting. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#partitioning, 2023. 4

[3] NVIDIA Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2023. 4

[4] NVIDIA Multi-Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html, 2023. 4

[5] Perlmutter supercomputer. https://docs.nersc.gov/systems/perlmutter/architecture/, 2023. 11

[6] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022. 13

[7] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023. 11

[8] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020. 13

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 1, 2, 10

[10] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *arXiv preprint arXiv:2310.18547*, 2023. 1, 3, 13

[11] Zhenqian Chen, Xinkui Zhao, Chen Zhi, and Jianwei Yin. Deepboot: Dynamic scheduling system for training and inference deep learning tasks in gpu cluster. *IEEE Transactions on Parallel and Distributed Systems*, 2023. 1

[12] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022. 4, 13

[13] Yujeong Choi and Minsoo Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233. IEEE, 2020. 13

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022. 1

[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017. 13

[16] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference, 2023. 3

[17] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023. 1

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 2

[19] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020. 4, 14

[20] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5(3):220–235, 2023. 3

[21] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 1

[22] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020. 1

[23] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *arXiv preprint arXiv:2205.11913*, 2022. 1

[24] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020. 13

[25] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1041–1057, 2022. 13

[26] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366*, 2021. 1, 3

[27] Robin J Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):1–16, 2014. 8

[28] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019. 1, 3

[29] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 1, 3, 14

[30] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Proceedings of Machine Learning and Systems 2020*, pages 497–511. 2020. 8

[31] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 11

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery. 1, 3

[33] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, 2018. 13

[34] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021. 1, 3

[35] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 835–850, 2023. 14

[36] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021. 1, 3

[37] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023. 13

[38] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022. 3

[39] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021. 1

[40] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 2023. 3

[41] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022. 1, 10, 11

[42] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen-tau Yih, and Madian Khabsa. Unipelt: A unified framework for parameter-efficient language model tuning. *arXiv preprint arXiv:2110.07577*, 2021. 3

[43] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023. 1

[44] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative large language model serving with speculative inference and token tree verification, 2023. 10

[45] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. *ASPLOS*, 2024. 13

[46] Deepak Narayanan, Keshav Santhanam, Peter Henderson, Rishi Bommasani, Tony Lee, and Percy Liang. Cheaply estimating inference efficiency metrics for autoregressive transformer models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. 10

[47] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. 1

[48] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. *arXiv preprint arXiv:2005.00247*, 2020. 1

[49] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. 2

[50] 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. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020. 1

[51] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021. 13

[52] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016. 3

[53] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022. 1

[54] Timo Schick and Hinrich Schütze. Exploiting cloze questions for few shot text classification and natural language inference. *arXiv preprint arXiv:2001.07676*, 2020. 3

[55] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud

provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020. 13

[56] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019. 13

[57] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023. 1, 3, 11, 13

[58] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023. 3

[59] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. 6

[60] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020. 1

[61] MosaicML NLP Team. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023. Accessed: 2023-05-05. 11

[62] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. 1, 10, 11

[63] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023. 1, 11

[64] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022. 5, 6, 7

[65] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax. https://github.com/huggingface/transformers, 2022. 10

[66] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023. 3

[67] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018. 4, 14

[68] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020. 4, 14

[69] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. A survey of multi-tenant deep learning inference on gpu. *arXiv preprint arXiv:2203.09040*, 2022. 4

[70] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022. 3, 9

[71] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019. 13

[72] Haoyu Zhang, Jianjun Xu, and Ji Wang. Pretraining-based natural language generation for text summarization. *arXiv preprint arXiv:1902.09243*, 2019. 1

[73] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association. 13

[74] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022. 10, 11

[75] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters. *arXiv preprint arXiv:2303.13803*, 2023. 4

[76] Han Zhou, Xingchen Wan, Ivan Vulić, and Anna Korhonen. Autopeft: Automatic configuration search for parameter-efficient fine-tuning. *arXiv preprint arXiv:2301.12132*, 2023. 3

[77] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. {PetS}: A unified framework for {Parameter-Efficient} transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504, 2022. 3, 13

[78] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 3