École Polytechnique Fédérale de Lausanne

Semester Project Report

Optimizations for Multi-LoRA Training and Fine-tuning

presented by

Viacheslav Surkov

Supervisors:

Prof. Dr. Anne-Marie Kermarrec

Milos Vujasinovic

Dr. Martijn De Vos

Dr. Rafael Pereira Pires

June, 2025

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

– DONALD KNUTH

Contents

Abstract

We present an optimized system for co-serving training and inference for Large Language Models using multiple Low-Rank Adaptation adapters. We analyze VRAM consumption across inference, full fine-tuning, and LoRA fine-tuning, and reveal that LoRA significantly narrows but does not eliminate the memory gap. To address the challenges of heterogeneous adapters and maximize throughput, we implement low-level CUDA optimizations using block-sparse matrix operations and introduce enhanced scheduling strategies, including an improved variant of Temporal Sharing. Our system is modular and model-agnostic, and requires no changes to the LLM architecture. This work demonstrates the feasibility of practical, high-performance multi-adapter serving on a single GPU and develops design insights for future LLM systems.

Introduction

In recent years, Large Language Models (LLMs) have been employed for a wide range of applications, including question answering [1], dialogue systems [2], and code generation [3]. Their architecture, typically a decoder-only variant of the Transformer model [4], is a well-established approach for implementation. However, some architectural adjustments have also attracted attention from the community—for example, DeepSeek's Multi-head Latent Attention [5] or the Mixture-of-Experts approach[6].

To tailor LLM generations to users' needs, one can fine-tune an LLM on a custom dataset that is much smaller than the pretraining data. However, fine-tuning typically requires a similar amount of GPU memory (VRAM) as full pretraining, which limits its accessibility. To alleviate this issue, researchers have proposed a variety of parameter-efficient fine-tuning (PEFT) methods, such as prefix tuning [7], prompt tuning [8], or adapters [9].

Low-Rank Adaptation (LoRA) [10] is a PEFT method that optimizes a low-rank additive component to weight matrices. Due to its memory efficiency, performance, and flexibility, it is one of the most widely adopted fine-tuning algorithms.

Numerous LLM providers—such as OpenAI, Claude, and Together.ai—offer services for fine-tuning LLMs, underscoring the importance of this topic. However, their infrastructure involves thousands of GPUs and a proprietary codebase that includes critical architectural decisions and optimizations for efficient

serving. This closed nature limits the adoption of similar services by academic institutions, small enterprises, and individual enthusiasts.

In this work, we aim to bridge this gap by implementing a system that serves an LLM alongside fine-tuning and inference of multiple LoRA modules—all on a single GPU—and by analyzing the impact of various optimizations.

Our contributions are:

- We integrate low-level sparse matrix optimizations introduced by Stanford STK [11] to improve multi-LoRA throughput.
- We analyze the key factors influencing LLM VRAM consumption, emphasizing the differences between inference and fine-tuning.
- Informed by this analysis, we introduce a novel enhanced temporal sharing scheduling scheme.
- Finally, we combine these ideas in an open-source implementation of a multi-LoRA fine-tuning and inference system.

We hope this work contributes to democratizing access to LLM fine-tuning and serving, enabling broader experimentation and innovation in the community.

Background

In this section, we introduce the foundational concepts relevant to our system: the architecture and behavior of Large Language Models (LLMs), the principle of Low-Rank Adaptation (LoRA), and the computational distinctions between fine-tuning and inference. This background motivates the need for a co-serving system and informs the design decisions presented in the subsequent chapters.

3.1 Large Language Models

Large Language Models (LLMs) are transformer-based architectures trained on vast corpora to perform a wide range of natural language tasks. Examples include GPT-3 [12], LLaMA [13], and Falcon [14]. These models are characterized by their depth (number of transformer blocks), width (hidden dimensions), and number of parameters, often reaching hundreds of billions.

During inference, LLMs generate text by predicting tokens one at a time based on a given prompt. This involves a forward pass through the model using cached key–value pairs for attention, reducing computational redundancy during autoregressive decoding. In contrast, training or fine-tuning requires both forward and backward passes and additional memory to store gradients and optimizer states.

LLMs are typically deployed in two modes:

- **Inference Mode:** Used for serving end-users. Models operate in no-

gradient mode, with weights frozen and intermediate tensors deallocated immediately after use.

- **Fine-Tuning Mode:** Used for adapting the base model to new tasks or domains. Weights may be updated partially or fully, and intermediate tensors are retained for gradient computation during backpropagation.

3.2 Low-Rank Adaptation

Low-Rank Adaptation (LoRA) is a technique designed to fine-tune large models efficiently by introducing trainable low-rank matrices into existing weight layers. Rather than updating all model weights, LoRA adds a pair of small matrices, $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{d \times r}$, where $r \ll d$, to key projection layers such as attention and feed-forward components.

The key properties of LoRA include:

- **Parameter Efficiency:** The number of trainable parameters is reduced significantly compared to full fine-tuning.

- **Modularity:** LoRA adapters can be stored, loaded, and composed independently of the base model.

- **Inference Compatibility:** Since LoRA layers are inserted additively into the forward pass, multiple adapters can be interchanged at runtime, enabling multi-task or user-personalized inference.

By decoupling task-specific learning from the full model, LoRA enables scalable fine-tuning and inference co-serving with lower memory and compute requirements.

3.3 Fine-Tuning and Inference: Computational Patterns

Fine-tuning and inference in LLMs exhibit distinct memory and compute behaviors, even though they share the same underlying model architecture. These differences are crucial when designing systems that aim to co-serve both workloads efficiently. In this section, we clarify the computational patterns involved in LLM inference, particularly its decomposition into prefill and decoding stages, and contrast them with fine-tuning behavior. We also describe the implications of using custom GPU kernels for performance optimization.

3.3.1 Inference: Prefill and Decoding Stages

Inference in autoregressive LLMs can be divided into two stages:

- **Prefill Stage:** The input prompt (sequence of tokens) is passed through the model in a standard forward pass. Intermediate outputs, such as keys and values from attention layers, are cached for reuse in subsequent steps.
- **Decoding Stage:** The model generates one token at a time, using previously cached key–value pairs. Only the most recent token is passed through the model, and the computation is significantly lighter than the prefill stage.

The prefill stage resembles standard inference, involving larger tensor computations, while the decoding stage is highly latency-sensitive and computationally sparse.

3.3.2 Fine-Tuning Behavior

Fine-tuning introduces significant overhead due to the backward pass. Unlike inference, it requires:

- Retention of intermediate activations for gradient computation during the backward pass.
- Additional memory for gradient and optimizer state storage.

When using LoRA, only a small subset of weights is updated, reducing the number of gradients and optimizer states that need to be tracked.

Understanding these differences is crucial when designing systems that serve both workloads simultaneously. Efficient co-serving requires balancing these competing demands while avoiding memory exhaustion or underutilization.

Core Idea

In this section, we outline the core ideas underlying the implementation of an optimized co-serving system for multi-LoRA LLM training and inference. First, we enumerate the functional requirements for the system. Second, we analyze VRAM consumption under different modes of LLM service. Insights from this analysis inform the architectural choices for the co-serving system. We also describe various optimization techniques used in the implementation, including CUDA-level optimizations and several query scheduling strategies.

4.1 Multi-LoRA System Requirements

The system should integrate the following components:

1. **Large Language Model**— a pretrained LLM that serves as the backbone for fine-tuning and inference using LoRA adapters.
2. **LoRA Adapter Set**— a dynamic collection of LoRA adapters. These adapters may have different ranks and can be added dynamically as they are fine-tuned.

The system is exposed to users via an HTTP API and must efficiently handle two types of queries:

1. `Generate(prompt, adapter_id)`— generates a completion for the given prompt using the LLM with the specified LoRA adapter `adapter_id`.

2. `Fine-Tune(corpus)`— fine-tunes a new LoRA adapter using the provided text corpus.

In general, we aim for predefined service-level objectives (SLOs) for latencies of `Generate` requests, and aspire to maximize throughput of `Fine-Tune` requests.

4.2 VRAM Analysis

Understanding the dynamics of VRAM consumption in LLMs under different serving modes (inference only, full fine-tuning, or LoRA fine-tuning) is essential for efficient system design. In particular, insights into these patterns can help predict VRAM usage in deployed systems, thereby reducing underutilization or the risk of memory overflows. In this section, we:

- Provide a brief overview of memory allocations and deallocations during no-grad forward passes, standard forward passes, and backward passes in LLMs.
- Analyze practical VRAM consumption and derive empirical relationships between batch size and VRAM usage under different workload types.

4.2.1 Forward and Backward Passes

When an LLM handles inference-only requests, the input tokens are passed through the transformer blocks without gradient computation—this is referred to as a no-grad forward pass. Each intermediate tensor is deallocated as soon as it is no longer needed.

During fine-tuning, the model undergoes two passes: a forward pass and a backward pass. Unlike inference, the forward pass in training stores auxiliary tensors (such as activations) that are later used by the backward pass to compute gradients with respect to the weights. This introduces additional

VRAM overhead: the forward pass retains these intermediate tensors, while the backward pass computes and stores gradients.

When using LoRA for fine-tuning, on one hand, we reduce the number of trainable weights, and consequently, the number of gradients to store. However, the forward pass still saves nearly all auxiliary tensors, as they are required for computing gradients of the adapter layers. Therefore, naively batching inference and fine-tuning requests together results in memory inefficiency.

4.2.2 VRAM consumption structure

For clarity, we break down the VRAM consumption during a single pair of forward and backward passes into four components:

1. M_{weights} — memory for model weights
2. $M_{\text{no\_grad}}$ — overhead from a no-grad forward pass (used for inference).
3. M_{act} — additional memory used in a training forward pass compared to a no-grad pass, due to tensors required for gradient computation
4. M_{backward} — additional memory used in the backward pass compared to the training forward pass. The primary component here is the gradients; however, since auxiliary tensors are deallocated during the backward pass, this component can shrink with increasing batch size.

Thus, the total VRAM consumption during inference is $M_{\text{weights}} + M_{\text{no\_grad}}$, while during fine-tuning it is $M_{\text{weights}} + M_{\text{no\_grad}} + M_{\text{act}} + M_{\text{backward}}$.

We compute the memory components by measuring 4 values:

1. VRAM occupied by model weights.
2. Peak VRAM consumption during no-grad forward pass: `with torch.no_grad(): model(batch)`
3. Peak VRAM consumption during gradient forward pass: `model(batch)`

4. Peak VRAM consumption during full forward-backward pass sequence: `logits=model(batch); logits.sum().backward()`

Then, $M_{\text{weights}} = (1)$, $M_{\text{no\_grad}} = (2) - (1)$, $M_{\text{act}} = (3) - (2)$, $M_{\text{backward}} = (4) - (3)$

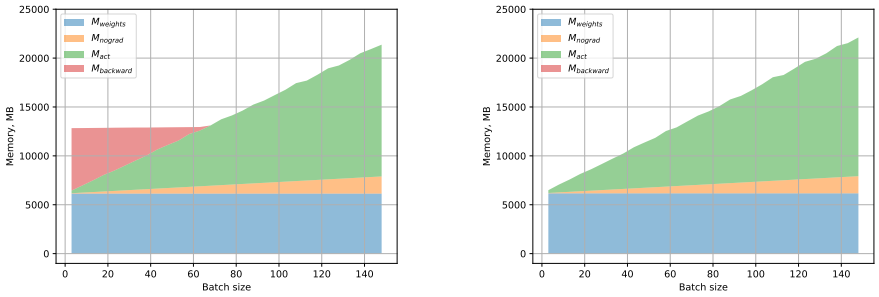
4.2.3 Empirical Analysis

The goal of this subsection is to estimate the VRAM consumption components of a real LLM. This evaluation helps us understand the differences between fine-tuning and inference modes, both with and without LoRA adapters, and provides intuition for making efficient implementation choices in the co-serving system.

The following configuration is used for the measurements in this section:

- Large Language Model: GPT-2 XL[12]
- Sequence length: 11 tokens
- Batch size: 3–1000
- VRAM measurements are based on PyTorch peak allocated memory. We observed that non-PyTorch GPU memory usage remains constant throughout the experiments.
- For the LoRA-enhanced model: `rank=32, target=["c_attn", "c_proj"]`

Figure 4.1 illustrates the relationship between batch size and memory components. While $M_{\text{no\_grad}}$ and M_{act} scale linearly with batch size, M_{backward} decreases and approaches zero around a batch size of ~ 70. Beyond this point, auxiliary tensors consume more memory than the gradients, and the backward pass contributes no additional overhead. Remarkably, LoRA only removes the M_{backward} component, thus, it is effective only on sufficiently small batch sizes, and its impact drops with increasing batch size.



(a) GPT-2 XL

(b) GPT-2 XL (frozen) + LoRA

Figure 4.1: Contribution of different components to LLM VRAM consumption. The total consumption is the sum of all components.

4.2.4 Discussion

LLM inference indeed requires significantly less VRAM than full fine-tuning. While LoRA reduces the gap, it does not eliminate it entirely: it primarily reduces the memory needed for the backward pass. For sufficiently large batch sizes, the VRAM consumption for both full and LoRA fine-tuning becomes comparable[1].

Therefore, naively merging inference and fine-tuning requests into a shared forward pass can result in unnecessary VRAM allocation (due to M_{act}), leading to inefficient GPU memory usage. Therefore, in the co-serving system, **we will process finetuning and inference batches separately**.

4.3 Low-Level Optimizations for Multi-LoRA Serving

4.3.1 Problem Formulation

We consider a system that performs forward passes on a batch of inputs using various LoRA adapters. Specifically, let R denote a request batch of length

[1] We exclude additional memory used by the optimizer. If optimizer states are also stored on the GPU, LoRA may still offer advantages due to the reduced number of trainable parameters.

BS, and let L denote a list of BS adapter identifiers. The system must perform a forward pass on each element of R, where each R_i is paired with the adapter L_i, which is attached to the LLM during the corresponding forward pass. The adapters may be heterogeneous—i.e., their ranks may vary—which makes batched execution more difficult. We refer to this challenge as the *multi-LoRA problem.*

Note that during fine-tuning or the inference prefill stage, each R_i represents a list of input tokens. However, during the inference decoding stage, R_i consists only of the last token, with the rest handled via the KV cache.

4.3.2 Naive Solution

A straightforward approach is to group requests R_i by their associated adapter ID and perform separate forward passes for each group. This allows each group to benefit from batched execution, thus leveraging parallelism within each group.

However, this strategy can lead to poor hardware utilization, especially when the number of distinct LoRA adapters is high, thereby limiting opportunities for parallelism across the full batch.

4.3.3 LoRA Grouping Solution

The naive approach can be improved by splitting the batch only at the point where the LoRA-specific A and B matrices are applied. This way, the backbone of the LLM remains fully batched and parallelized, preserving more of the available performance potential.

This approach is used in the Hugging Face PEFT library for multi-LoRA serving.

4.3.4 Megablocks-Enhanced Approach

The optimized method builds on the idea that multi-LoRA operations can be formulated as a series of block-sparse matrix multiplications. (See Figure 4.2

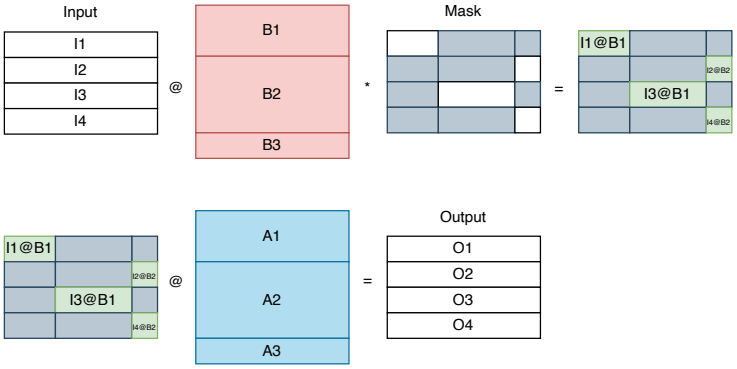


Figure 4.2: Illustration of multi-LoRA computations using DDS and SDD operations

for visual illustrations.) Two types of block-sparse operations are required:

1. **DDS (Dense–Dense–Sparse):** Takes $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times k}$—the dense matrices to be multiplied—and a block-sparse mask $M \in \{0,1\}^{n \times k}$. It returns $(AB) \odot M$, where $\odot$ denotes element-wise multiplication, resulting in a block-sparse output.

2. **SDD (Sparse–Dense–Dense):** Takes S—a block-sparse matrix –and a dense matrix B, and returns their product.

If the block dimensions are divisible by 16, these operations can be efficiently parallelized on GPU hardware, avoiding unnecessary computation. Optimized CUDA kernels for these operations are introduced by Stanford STK [11].

Implementation Notes. Both SDD and DDS require block dimensions to be divisible by 16, as lower sizes are not supported by Triton. For fine-tuning and inference prefill stages, this is generally not an issue–padding can be applied with minimal overhead. However, in the decoding stage, where LoRA operates on individual tokens, padding increases the tensor size by a factor of 16. Despite this, the additional overhead is relatively minor compared to the cost of applying the linear layer to which the LoRA adapter is attached.

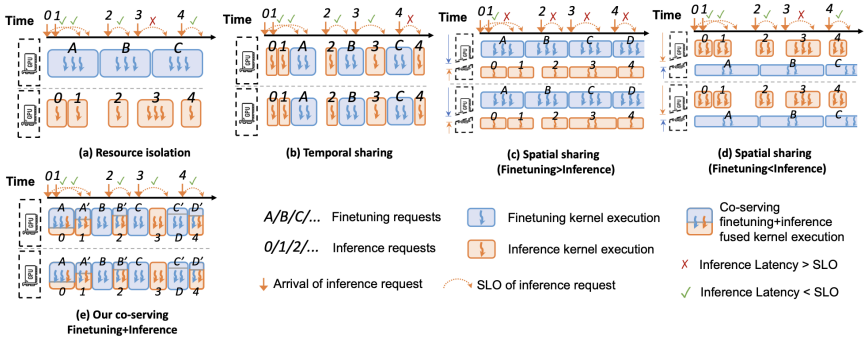


Figure 4.3: Overview of fine-tuning & inference scheduling approaches. Borrowed from FlexLLM [15]. "Our co-serving Finetuning+Inference" refers to FlexLLM's approach

4.4 Fine-Tuning & Inference Scheduling Strategies

4.4.1 Overview of Scheduling Approaches

The FlexLLM study [15] identifies four types of strategies for scheduling fine-tuning and inference requests:

1. **Resource Isolation.** Fine-tuning and inference requests are isolated from each other and executed on separate GPU instances. This approach naturally prevents race conditions and ensures high performance due to the absence of interference. However, it requires allocating video memory for the model weights twice.

2. **Temporal Sharing.** This method allows fine-tuning and inference requests to access GPU resources in turns, thereby reducing GPU memory usage. However, since request arrival times are unpredictable, this can lead to significant execution delays. For example, if an inference request arrives in the middle of a lengthy fine-tuning session, it may be queued for a long time. The authors propose interleaving one fine-tuning request with n inference requests to mitigate this issue. Nevertheless, if fine-tuning

takes several minutes, inference requests may still experience substantial delays.

3. **Spatial Sharing.** This approach assigns specific portions of GPU memory to both fine-tuning and inference requests, enabling simultaneous execution. It also avoids duplicating model weights. However, it lacks flexibility in dynamic memory allocation and depends on advanced GPU features such as Multi-Process Service, which are only available on a limited set of NVIDIA GPUs.

4. **FlexLLM Co-Serving Approach.** Designed to overcome the drawbacks of the previous methods, this strategy integrates fine-tuning and inference kernels for simultaneous batched execution of both types of requests.

Figure 4.3 depicts the GPU execution timelines associated with scheduling approaches above.

The authors of FlexLLM evaluated these approaches under synthetic workloads. Their results suggest that *Temporal Sharing* achieved the best throughput, though it required careful tuning of the n parameter. As discussed in Sec. 4.2, VRAM consumptions under fine-tuning and inference modes are significantly different even when LoRA is applied, therefore approaches with batch fusing will suffer either from VRAM overutilization, or from overhead caused by custom kernels, which are less optimized than built-in tools. Thus, the efficiency of *temporal sharing* compared to the other approaches is in agreement with our previous analysis.

4.4.2 Enhanced Temporal Sharing

The *Temporal Sharing* approach can result in significant delays and queuing of inference requests when fine-tuning takes considerable time. For instance, if inference requests take 100ms to complete but fine-tuning requires four minutes, inference requests may have to wait for at least two minutes on average,

regardless of how large the n parameter is.

To address this issue, we propose "pausing" the fine-tuning execution in favor of incoming inference requests. Specifically, if there are already B inference requests waiting, or if the first inference request in the queue has been waiting for more than t milliseconds, the system waits until the current fine-tuning batch completes, and then processes the batch of inference requests. Once generation is finished, fine-tuning resumes.

We refer to this method as *Enhanced Temporal Sharing (ETS)*. It keeps the advantages of *Temporal Sharing*—such as storing the model weights only once—and it also mitigates delays for inference requests.

4.4.3 Implementation details of ETS

To implement *Enhanced Temporal Sharing*, we maintain inference and fine-tuning request queues. While inference queue stores individual inference requests, we split fine-tune requests into batches and store them in the fine-tune queue. The users' API calls trigger inserting requests into the queues. The serving loop checks if requests are ready for dispatching and gives priority to inference requests. Refer to Algorithm 7 for details. This approach ensures that inference requests will not be blocked by lengthy fine-tune requests, as we process them in separate batches.

Algorithm 1 Enhanced Temporal Sharing

Data: Async queues: `inference_queue`, `finetune_queue`
Data: Constants: `INFERENCE_BATCH_SIZE`, `INFERENCE_DELAY`
while *True* **do**
  if *inference_queue.size()* $\geq$ *INFERENCE_BATCH_SIZE* **or** *current_time* $-$ *first_arrival* $>$ *INFERENCE_DELAY* **then**
    Take `INFERENCE_BATCH_SIZE` requests from `inference_queue`
    Dispatch batch for inference Perform the inference request (synchronously)
  else if *finetune_queue is not empty* **then**
    Take a batch from `finetune_queue`
    Dispatch batch for fine-tuning
    Perform the fine-tuning request (synchronously)
  else
    Wait until at least one of the conditions above is met

Evaluation

Since low-level optimizations and scheduling are orthogonal, we divide this chapter into two parts. In the first part, we compare different approaches to the multi-LoRA problem, evaluating downstream inference and fine-tuning throughput on a predefined text collection. In the second part, we describe the synthetic workload generation and compare scheduling strategies under this workload.

5.1 Low-Level Optimizations

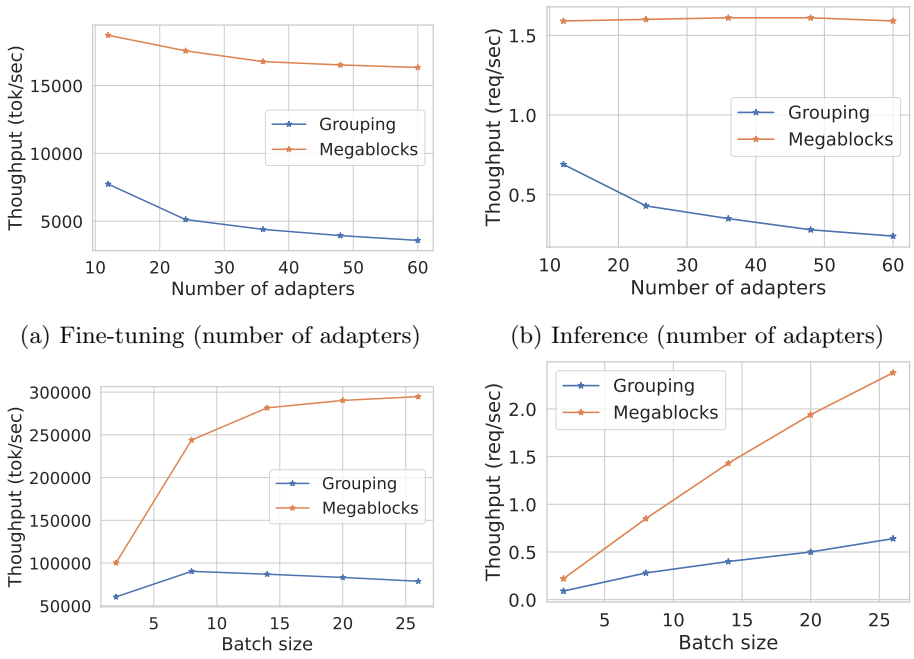
We evaluated three multi-LoRA approaches—naive, grouping, and megablocks-enhanced—on both fine-tuning and inference tasks. The fine-tuning task uses three datasets[1][2][3], each assigned to one of LoRA adapter (but many LoRA adapters can be trained on the same dataset). The inference task utilizes the *Awesome GPT Prompts* dataset[4], where prompts are completed using multiple different LoRA adapters. Within a batch, adapters are assigned independently and uniformly at random. For both fine-tuning and inference, we measure throughput in tokens per second.

[1] https://huggingface.co/datasets/mlabonne/FineTome-100k
[2] https://huggingface.co/datasets/bitext/Bitext-customer-support-llm-chatbot-training-dataset
[3] https://huggingface.co/datasets/mlabonne/guanaco-llama2
[4] https://github.com/f/awesome-chatgpt-prompts

5.1.1 Experimental Setup

1. **Large Language Model:** GPT-2 Medium (bfloat16)
2. **LoRA Adapters:** 1/4 of adapters have ranks of 16, 32, 64, or 128; `target=["c_attn", "c_proj"]`
3. **Default Batch Size:** 16; **Default Number of Adapters:** 24; **Maximum Sequence Length:** 512
4. **Hardware:** NVIDIA A100 GPU (80 GB VRAM), AMD EPYC 7543 32-Core Processor

5.1.2 Results



(a) Fine-tuning (number of adapters)

(b) Inference (number of adapters)

(c) Fine-tuning (batch size)

(d) Inference (batch size)

Figure 5.1: Throughput evaluation results for fine-tuning and inference across different numbers of adapters and batch sizes.

We compare throughputs of LoRA grouping and megablocks-enhanced approaches. The evaluation results are shown in Figure 5.1. The megablocks-

enhanced approach outperforms the grouping method in both inference and fine-tuning tasks. Moreover, its throughput remains stable as the number of adapters increases, unlike the grouping method, which exhibits significant degradation. These results suggest that the megablocks-enhanced approach scales effectively with larger numbers of adapters and larger batch sizes.

5.2 Scheduling

5.2.1 Synthetic workload

We generate concurrent inference and fine-tuning workloads. To synthesize the inference workloads we sample from *Awesome GPT Prompt* dataset and send the request to the system every $T_{inference}$ seconds. For fine-tuning workload, we sample up to $N_{fine-tuning}$ requests similarly as in Sec. 5.1 every $T_{fine-tuning}$ seconds. The requests adapters are sampled among 32 adapters of rank 32. See the default values of these parameters in Table 5.1.

$T_{\text{inference}}$	$T_{\text{fine-tune}}$	$N_{\text{fine-tune}}$
0.1	20	512

Table 5.1: Default parameters for synthetic workloads

5.2.2 Systems' settings

To implement the systems for evaluation, we use Megablock-enhanced approach for multi-LoRA problem. Fine-tuning requests are split into batches of size 32, while the inference requests are grouped into bathes of size up to 64. We don't send an inference batch for execution, until its size is 64 or 5 seconds have passed since the arrival of the first request in the batch.

5.2.3 Metrics

1. **Inference latency.** The time difference between the arrival of a request and its completion.

2. **Inference Service-level objective (SLO) attainment.** The ratio of inference requests completed within the SLO (10 seconds or 1 minute).

3. **Fine-tuning throughput.** The number of fine-tuning tokens processed per second.

5.2.4 Results

The evaluation results are shown in Table 5.1. *Enhanced Temporal Sharing* (ETS) maintains a comparable fine-tuning throughput while keeping all inference latencies within 10 seconds. This is possible because ETS allows fine-tuning execution to be paused in favor of incoming inference requests. The advantage becomes even more apparent in Fig. 5.2, where increasing the size of fine-tuning batches severely degrades inference latency under Temporal Sharing (TS), while ETS maintains consistently low latencies. These results highlight ETS as a practical and effective scheduling strategy to balance responsiveness and throughput in mixed LLM workloads.

	ETS	TS (5)	TS (10)	TS (20)
Inference latency, mean (s)	**6.18**	38.98	12.96	12.85
SLO: 10s	**100%**	0.00%	24.76%	26.12%
SLO: 1m	**100%**	83.08%	97.37%	96.12%
Fine-tuning throughput	9071	12324	9852	9480

Table 5.2: Evaluation results for *Enhanced Temporal Sharing* (ETS) and *Temporal Sharing* (TS) approach. Interleave parameter N is put inside parentheses

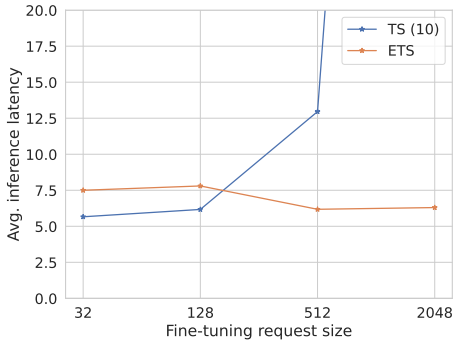


Figure 5.2: Relation between average inference latency and size of fine-tuning requests

Discussion and Future Work

In this section, we provide a discussion of the developed multi-LoRA system, outline its limitations, and suggest prospective directions for future improvements.

Discussion. Our system employs low-level optimizations to accelerate multi-LoRA serving, along with enhanced scheduling mechanisms for co-serving inference and fine-tuning workloads. The VRAM analysis confirmed our hypothesis that fine-tuning consumes significantly more GPU memory than inference. However, a surprising observation was that LoRA adapters bridge only part of this memory gap, and the difference becomes negligible at sufficiently large batch sizes. Therefore, LoRA offers advantages primarily at small batch sizes. Unlike many prior works, we aimed to keep our implementation as modular as possible. For instance, GPT-2 can be easily swapped out for other LLMs, as we made no modifications to the model's core implementation.

Limitations. Although the Megablocks-enhanced approach can be adapted to a wide range of LLMs, it may sometimes require additional engineering effort. For example, if the underlying LLM kernels fuse target matrix operations with other computations, block-sparse matrix operations must be integrated into those fused kernels. Additionally, the method requires block dimensions to be divisible by 16, which prevents the use of LoRA adapters with rank 8—a commonly adopted configuration in the community.

Future Directions. Further gains in low-level performance could be achieved

by fusing block-sparse matrix operations directly into existing LoRA logic. On the scheduling side, improvements may come from more advanced inference techniques. For example, ORCA's iteration scheduling [16] or the inference optimizations introduced in DeepSpeed FastGen [17] could significantly improve overall system performance.

The current implementation of the system lacks features such as sophisticated parameterization of fine-tuning and inference requests — for example, the use of advanced optimizers, learning rate schedulers, hyperparameter tuning, or specifying generation parameters like temperature, top-k, and maximum length. These issues can be addressed through additional engineering, as the system architecture does not impose any fundamental restrictions on such extensions.

Related Work

In this section, we discuss relevant prior research.

Efficient Large Language Model Serving is a rapidly developing research direction that aims to optimize VRAM usage and inference latency, thereby significantly reducing serving costs and improving user experience. The ORCA system [18] employs iteration-level scheduling and selective batching to merge tensors corresponding to queries of different lengths and execution phases. Subsequent work [19, 17] enhanced this approach through the SplitFuse technique, which chunks and fuses prompts of varying lengths to ensure uniform forward pass sizes. Our serving strategy and optimizations are orthogonal to these techniques and can therefore be seamlessly integrated for future enhancements.

Multiple LoRA Serving has been extensively explored in recent studies. Punica [16] introduces a custom CUDA kernel to batch GPU operations across different LoRA models. S-LoRA [20] improves multi-LoRA inference by using unified paging for adapter weights, custom CUDA kernels, and efficient parallel communication strategies. Flex-LLM [15] applies multiple optimizations such as dependent parallelization and graph pruning to improve co-serving and both LLM inference and PEFT fine-tuning. However, these methods tend to be sensitive to system configurations, and adapting them to a broader range of LLMs requires substantial engineering effort. In contrast, our work introduces a co-serving system that easily generalizes to a wide range of LLMs.

Custom CUDA Kernels for LLMs. Several systems have introduced highly optimized CUDA kernels to accelerate dense linear algebra operations. For instance, NVIDIA's CUTLASS and TensorRT-LLM provide hand-tuned kernels for matrix multiplication and attention. FlashAttention [21] introduced a memory-efficient attention mechanism using custom CUDA kernels that avoid materializing large intermediate tensors, significantly reducing memory bandwidth usage and improving transformer performance. Our work leverages custom CUDA kernels for sparse-block matrix multiplication, as proposed by [11], to efficiently handle multiple heterogeneous LoRA adapters within a single batch.

Conclusion

This study proposes an optimized co-serving system for training and inference using multiple LoRA adapters on large language models.

Our main contributions include (1) a VRAM characterization of different LLM execution modes: no-grad inference, full fine-tuning, and LoRA-based fine-tuning; (2) the development of a multi-LoRA serving strategy using block-sparse matrix operations via Megablocks; and (3) Enhanced Temporal Sharing — a scheduling strategy — to balance inference latency and fine-tuning throughput in a shared environment.

These results demonstrate that co-serving fine-tuning and inference is feasible without duplicating model weights or significantly compromising performance. Furthermore, our findings on VRAM scaling suggest practical batch size thresholds for when LoRA remains memory-efficient. The proposed kernel-level and scheduling optimizations may serve as foundational techniques for future adapter-based serving systems.

As LLMs are being adopted to an increasing number of use cases, support for low-cost customization via adapters will become increasingly important. Systems that can co-serve heterogeneous workloads with minimal overhead will be crucial for democratizing LLM deployment.

Our system paves the way for efficient, scalable, and modular adapter-based training and inference, laying the groundwork for more resource-efficient LLM-serving infrastructures.

Bibliography

[1] M. Yue, "A survey of large language model agents for question answering," 2025. [Online]. Available: https://arxiv.org/abs/2503.19213

[2] H. Wang, L. Wang, Y. Du, L. Chen, J. Zhou, Y. Wang, and K.-F. Wong, "A survey of the evolution of language model-based dialogue systems," 2023. [Online]. Available: https://arxiv.org/abs/2311.16789

[3] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," Aug. 2023, arXiv:1706.03762 [cs]. [Online]. Available: http://arxiv.org/abs/1706.03762

[5] DeepSeek-AI, A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Xu, H. Yang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Chen, J. Yuan, J. Qiu, J. Song, K. Dong, K. Gao, K. Guan, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Zhang, M. Li, M. Wang, M. Zhang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin,

R. Ge, R. Pan, R. Xu, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Zheng, T. Wang, T. Pei, T. Yuan, T. Sun, W. L. Xiao, W. Zeng, W. An, W. Liu, W. Liang, W. Gao, W. Zhang, X. Q. Li, X. Jin, X. Wang, X. Bi, X. Liu, X. Wang, X. Shen, X. Chen, X. Chen, X. Nie, X. Sun, X. Wang, X. Liu, X. Xie, X. Yu, X. Song, X. Zhou, X. Yang, X. Lu, X. Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zhao, Y. Sun, Y. Li, Y. Wang, Y. Zheng, Y. Zhang, Y. Xiong, Y. Zhao, Y. He, Y. Tang, Y. Piao, Y. Dong, Y. Tan, Y. Liu, Y. Wang, Y. Guo, Y. Zhu, Y. Wang, Y. Zou, Y. Zha, Y. Ma, Y. Yan, Y. You, Y. Liu, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Huang, Z. Zhang, Z. Xie, Z. Hao, Z. Shao, Z. Wen, Z. Xu, Z. Zhang, Z. Li, Z. Wang, Z. Gu, Z. Li, and Z. Xie, "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model," Jun. 2024, arXiv:2405.04434 [cs]. [Online]. Available: http://arxiv.org/abs/2405.04434

[6] W. Cai, J. Jiang, F. Wang, J. Tang, S. Kim, and J. Huang, "A Survey on Mixture of Experts in Large Language Models," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–20, 2025, arXiv:2407.06204 [cs]. [Online]. Available: http://arxiv.org/abs/2407.06204

[7] X. L. Li and P. Liang, "Prefix-Tuning: Optimizing Continuous Prompts for Generation," Jan. 2021, arXiv:2101.00190 [cs]. [Online]. Available: http://arxiv.org/abs/2101.00190

[8] B. Lester, R. Al-Rfou, and N. Constant, "The Power of Scale for Parameter-Efficient Prompt Tuning," Sep. 2021, arXiv:2104.08691 [cs]. [Online]. Available: http://arxiv.org/abs/2104.08691

[9] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. d. Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-Efficient Transfer Learning for NLP," Jun. 2019, arXiv:1902.00751 [cs]. [Online]. Available: http://arxiv.org/abs/1902.00751

[10] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," Oct. 2021, arXiv:2106.09685 [cs]. [Online]. Available: http://arxiv.org/abs/2106.09685

[11] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "MegaBlocks: Efficient Sparse Training with Mixture-of-Experts," Nov. 2022, arXiv:2211.15841 [cs]. [Online]. Available: http://arxiv.org/abs/2211.15841

[12] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," Jul. 2020, arXiv:2005.14165 [cs]. [Online]. Available: http://arxiv.org/abs/2005.14165

[13] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," Feb. 2023, arXiv:2302.13971 [cs]. [Online]. Available: http://arxiv.org/abs/2302.13971

[14] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, E. Goffinet, D. Hesslow, J. Launay, Q. Malartic, D. Mazzotta, B. Noune, B. Pannier, and G. Penedo, "The Falcon Series of Open Language Models," Nov. 2023, arXiv:2311.16867 [cs]. [Online]. Available: http://arxiv.org/abs/2311.16867

[15] G. Oliaro, X. Miao, X. Cheng, V. Kada, R. Gao, Y. Huang, R. Delacourt, A. Yang, Y. Wang, M. Wu, C. Unger, and Z. Jia, "FlexLLM: A System for Co-Serving Large Language Model Inference

and Parameter-Efficient Finetuning," May 2025, arXiv:2402.18789 [cs]. [Online]. Available: http://arxiv.org/abs/2402.18789

[16] L. Chen, Z. Ye, Y. Wu, D. Zhuo, L. Ceze, and A. Krishnamurthy, "Punica: Multi-Tenant LoRA Serving," Oct. 2023, arXiv:2310.18547 [cs]. [Online]. Available: http://arxiv.org/abs/2310.18547

[17] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko, and Y. He, "DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference," Jan. 2024, arXiv:2401.08671 [cs]. [Online]. Available: http://arxiv.org/abs/2401.08671

[18] USENIX Association, Ed., *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation: July 11-13, 2022, Carlsbad, CA, USA*. Berkeley, CA: USENIX Association, 2022, meeting Name: USENIX Symposium on Operating Systems Design and Implementation.

[19] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills," Aug. 2023, arXiv:2308.16369 [cs]. [Online]. Available: http://arxiv.org/abs/2308.16369

[20] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer, J. E. Gonzalez, and I. Stoica, "S-LoRA: Serving Thousands of Concurrent LoRA Adapters," Jun. 2024, arXiv:2311.03285 [cs]. [Online]. Available: http://arxiv.org/abs/2311.03285

[21] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," Jun. 2022, arXiv:2205.14135 [cs]. [Online]. Available: http://arxiv.org/abs/2205.14135