LoRAFusion: Efficient LoRA Fine-Tuning for LLMs

Zhanda Zhu
University of Toronto, Vector Institute, NVIDIA
zhanda.zhu@mail.utoronto.ca

Qidong Su
University of Toronto, Vector Institute, NVIDIA
qdsu@cs.toronto.edu

Yaoyao Ding
University of Toronto, Vector Institute, NVIDIA
dingyaoyao.cs@gmail.com

Kevin Song
University of Toronto, Vector Institute, NVIDIA
xinyang.song@utoronto.ca

Shang Wang
University of Toronto, Vector Institute, NVIDIA
wangsh46@cs.toronto.edu

Gennady Pekhimenko
University of Toronto, Vector Institute, NVIDIA
pekhimenko@cs.toronto.edu

arXiv:2510.00206v1 [cs.LG] 30 Sep 2025

Abstract

Low-Rank Adaptation (LoRA) has become the leading Parameter-Efficient Fine-Tuning (PEFT) method for Large Language Models (LLMs), as it significantly reduces GPU memory usage while maintaining competitive fine-tuned model quality on downstream tasks. However, existing LLM LoRA fine-tuning systems mainly reuse optimizations from traditional full-model fine-tuning, and therefore cannot take full advantage of LoRA's unique characteristics. We identify two key inefficiencies of existing works. First, existing LoRA fine-tuning systems incur substantial runtime overhead due to redundant memory accesses on large activation tensors. Second, they miss the opportunity to concurrently fine-tune multiple independent LoRA adapters that share the same base model on the same set of GPUs. This leads to missed performance gains such as reduced pipeline bubbles, better communication overlap, and improved GPU load balance.

To address these issues, we introduce LoRAFusion, an efficient LoRA fine-tuning system for LLMs. At the kernel level, we propose a graph-splitting method that fuses memory-bound operations. This design eliminates unnecessary memory accesses and preserves the performance of compute-bound GEMMs without incurring the cost of recomputation or synchronization. At the scheduling level, LoRAFusion introduces an adaptive batching algorithm for multi-job fine-tuning. It first splits LoRA adapters into groups to intentionally stagger batch execution across jobs, and then solves a bin-packing problem within each group to generate balanced, dependency-aware microbatches. LoRAFusion achieves up to 1.96× (1.47× on average) end-to-end speedup compared to Megatron-LM, and up to 1.46× (1.29× on average) improvement over mLoRA, the state-of-the-art multi-LoRA fine-tuning system. Our fused kernel achieves up to 1.39× (1.27× on average) kernel performance improvement and can directly serve as a plug-and-play replacement in existing LoRA systems. We open-source LoRAFusion at https://github.com/CentML/lorafusion.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**; **Distributed computing methodologies**; **Machine learning**.

Keywords: LLM, LoRA, Systems for Machine Learning, Kernel Fusion, Distributed training

ACM Reference Format:
Zhanda Zhu, Qidong Su, Yaoyao Ding, Kevin Song, Shang Wang, and Gennady Pekhimenko. 2026. LoRAFusion: Efficient LoRA Fine-Tuning for LLMs. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3767295.3769331

1 Introduction

Pre-trained Large Language Models (LLMs), such as GPT [2] and LLaMa [86], have demonstrated strong capabilities across diverse tasks, including text generation [8, 103], question answering [29, 40], and code generation [10, 64]. To adapt these models to personalized or domain-specific tasks, *fine-tuning* on pre-trained LLMs is typically performed. Such adaptation is essential for scenarios like biomedical analysis [83, 101], personalized chatbot interactions [92], or specialized customer support [99]. However, traditional full-model fine-tuning, where all model parameters are learned, requires substantial hardware resources such as multiple nodes, each node equipped with multiple flagship GPUs (e.g., NVIDIA B200 GPUs [68]) and interconnected by high-speed links (e.g., NVLink [67] and InfiniBand [66]). For instance, full-model fine-tuning of LLaMa-3.1-70B [56] requires approximately 1120GB of GPU memory for model states alone (parameters, gradients, optimizer states), making this approach prohibitively expensive for practical applications [58].

To mitigate the substantial hardware requirements, Parameter-Efficient Fine-Tuning (PEFT) methods [14, 16, 30, 32, 37, 39, 44, 45, 48, 52, 76, 100, 102] have emerged. These approaches significantly reduce resource usage by

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769331

keeping pre-trained LLM parameters *frozen* (not updated during fine-tuning) and selectively updating only a small set of injected trainable parameters, known as *adapters*. Among these methods, Low-Rank Adaptation (LoRA) [32] and its variants [14, 16, 45, 52] are particularly popular due to their simplicity and effectiveness. LoRA freezes the model's pre-trained weights and adds a residual branch composed of two trainable low-rank linear layers: a down-projection from the input dimension k to a smaller rank r, followed by an up-projection back to the output dimension n, where $r \ll min(n, k)$. This low-rank branch is added to the output of the original frozen layer, enabling task-specific adaptation without modifying the base model (see Figure 1 and Section 2.1). For example, fine-tuning LLaMa-3.1-70B [56] using LoRA adapters with a low-rank dimension of 16 introduces only 0.29% additional parameters, reducing GPU memory usage to 142GB while preserving strong model quality [14, 100].

Due to these substantial resource savings, LoRA fine-tuning has become widely adopted on both cloud platforms [25, 71, 77, 85] and local environments [57, 108]. To realistically apply LoRA in practice and fine-tune high-quality adapters, users often run multiple jobs in parallel. These jobs may explore different hyperparameter settings [87] or continuously adapt models to evolving datasets and user needs [34, 36]. As a result, fine-tuning throughput, measured as the number of training samples processed per second, has become a key metric for reducing both cost and overall training time.

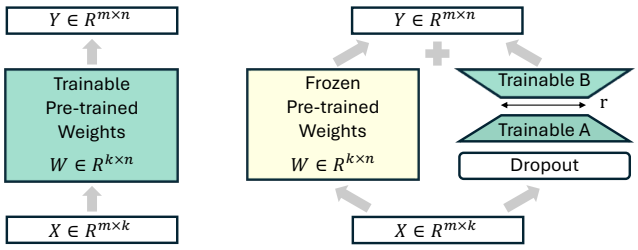
However, despite significant algorithmic advances in LoRA-based approaches [14, 37, 52] and explorations in serving scenarios [9, 79, 94, 110], existing LoRA fine-tuning systems for LLMs largely reuse optimizations designed for full-model fine-tuning. Specifically, systems like PEFT Library [55], LLaMA-Factory [108], and llama-cookbook [57] typically rely on a subset of parallelization methods such as Fully Sharded Data Parallelism (FSDP) [73, 104], Tensor Parallelism (TP) [53, 81], or Pipeline Parallelism (PP) [33, 46, 62] to fit the training into GPU memory and achieve efficiency with multiple GPUs. While such techniques are still useful, our analysis reveals they do not sufficiently address the unique characteristics of LoRA fine-tuning, causing significant inefficiencies.

The first limitation is the high runtime overhead introduced by LoRA adapters. LoRA adapters add less than 1% parameters compared to the full model and are thus expected to incur minimal computation and memory overhead. However, our profiling shows that applying LoRA reduces training throughput by approximately 40% compared to the original frozen model. This overhead comes from increased memory traffic: the small LoRA projection layers are memory-bandwidth-bound due to their low-rank dimensions, and the operations in the adapter repeatedly load and store large activation tensors. These factors together increase global memory access by up to 2.64×, as detailed in Section 3.1.

The second limitation is the lack of efficient support for joint fine-tuning across multiple LoRA adapters. Typically, each adapter is fine-tuned independently, even if they share the same base model. Since LoRA adapters are lightweight and add minimal memory footprint pressure, multiple jobs can be combined and run jointly on the same GPUs. We refer to this strategy as *multi-LoRA fine-tuning* [98], which is highly practical for hyperparameter tuning [87] and multi-tenant cloud services [25, 85]. Although multi-LoRA optimization [9, 79, 94] has been widely used in LLM serving, the motivation in our setting is entirely different. Serving systems batch requests to increase arithmetic intensity during autoregressive single-token decoding [9, 79], whereas fine-tuning already processes full sequences with sufficient arithmetic intensity. Instead, the primary benefit of multi-LoRA fine-tuning comes from mitigating the overhead of distributed parallelism [98].

While existing multi-LoRA fine-tuning systems like mLoRA [98] can reduce pipeline bubbles by filling them with independent groups of samples from different adapters, we observe that this approach is incomplete and still suffers from significant inefficiencies. First, it still relies on generic LoRA kernels that are bottlenecked by redundant memory accesses to large activation tensors as discussed previously, and batching more samples does not help. Second, it fails to address the load imbalance across GPUs. Realistic fine-tuning workloads often contain samples with variable sequence lengths, leading to imbalanced work across GPUs. This imbalance creates idle time in data-parallel replicas and increases pipeline bubbles from poorly aligned microbatches. As we analyze in Section 3.2, strategically grouping and scheduling samples across multiple jobs is critical to mitigate this imbalance and unlock further efficiency gains.

Based on these insights, we argue that an efficient LoRA fine-tuning system must both reduce the runtime overhead of LoRA adapters and leverage multi-LoRA optimization opportunities to reduce distributed training overhead. To meet these requirements, we propose *LoRAFusion*, an efficient multi-level fusion system tailored specifically for LoRA fine-tuning of LLMs. At the kernel level, our key insight is that most of LoRA's overhead comes from memory-bandwidth-bound operations on large activation tensors. We address this by splitting the computation graph at the point where the tensor size shrinks to the low-rank dimension r. This allows us to fuse memory-bandwidth-bound operations without recomputation or synchronization, while preserving the optimal performance of compute-bound matrix multiplications. This design leads to our `FusedLoRA` and `FusedMultiLoRA` kernels. At the scheduling level, LoRAFusion enables concurrent fine-tuning of multiple LoRA adapters that share the same base model by batching samples across jobs. Compared to the existing multi-LoRA systems, this further improves system throughput by reducing pipeline bubbles and balancing GPU workloads. The challenge is that such batching must



(a) Full Fine-tuning (b) LoRA Fine-tuning

Figure 1. Comparison between traditional full model fine-tuning and LoRA fine-tuning.

preserve execution dependencies between global batches, especially under pipeline parallelism. To address this, we first group adapters in a way that creates natural gaps between their batches, then solve a bin-packing problem within each group to construct balanced, dependency-safe microbatches using a combination of Mixed Integer Linear Programming (MILP) and greedy heuristics.

We implement LoRAFusion on top of Megatron-LM [81], a state-of-the-art distributed training framework, to support efficient parallelism and scalability. We then extensively evaluate LoRAFusion on a wide range of LLMs, including LLaMa-3.1-8B, Qwen-2.5-32B, and LLaMa-3.1-70B, across NVIDIA H100 and L40S GPUs with various realistic datasets. Results show that LoRAFusion achieves up to 1.96× (1.47× on average) end-to-end speedup compared to Megatron-LM with FSDP and PP, and up to 1.46× (1.29× on average) improvement over mLoRA [98], the state-of-the-art multi-LoRA fine-tuning system. Additionally, our fused kernel alone achieves up to 1.39× (1.27× on average) speedup compared to the default LoRA implementation on NVIDIA H100 GPUs, and can directly serve as a plug-in replacement in existing systems, offering immediate benefits to the broader community.

Overall, this paper makes the following contributions:

- We identify two key limitations in existing LoRA fine-tuning systems: high runtime overhead from redundant memory access, and missed opportunities for optimizing multi-job training. To address both, we propose LoRAFusion, a multi-level fusion system for accelerating LLM fine-tuning on modern GPU clusters.
- We propose a horizontal fusion strategy that reduces redundant memory access without disrupting compute-bound performance. We also design a scheduling algorithm that groups adapters across fine-tuning jobs and batches their samples to improve GPU load balance and reduce pipeline overhead.
- We evaluate LoRAFusion across diverse LLMs, datasets, and GPU platforms, showing significant improvements over existing LoRA fine-tuning systems. Our fused kernel also provides strong standalone gains and can be directly integrated into more general LoRA systems.

Table 1. Notation for the LoRA fine-tuning in this paper.

Symbol	Description
r	LoRA rank
m	Number of tokens (batch size × seq length)
k, n	Input/Output dimension of the weight matrix
W	Base model weights. Size: (k, n).
A	First LoRA weights. Size: (k, r)
B	Second LoRA weights. Size: (r, n)
X	Input tensor. Size: (m, k)
$\widehat{X}$	Input tensor after dropout. Size: (m, k)
Y	Output tensor. Size: (m, n)

2 Background

We first provide an introduction to LoRA fine-tuning (§2.1), and then present an overview of current system-level optimizations for fine-tuning (§2.2).

2.1 LLM LoRA Fine-tuning

Training an LLM from scratch demands substantial amounts of data, millions of GPU hours, and significant costs [58]. Fine-tuning pre-trained LLMs, such as LLaMa [20, 56] and Qwen [6, 97] is thus more practical. Fine-tuning preserves pre-trained capabilities while adapting the model to specialized downstream tasks [15]. While fine-tuning reduces the data needs and training iterations, traditional full-model fine-tuning still requires similarly large GPU memory, due to the vast number of trainable parameters.

PEFT methods [32, 39, 44] address the memory requirements by freezing pre-trained parameters (keeping them non-updatable) and only training a small set of newly introduced parameters called *adapters*. Among these methods, LoRA [32] is currently most widely used due to its simplicity and effectiveness. As illustrated in Figure 1, LoRA injects two small trainable matrices alongside each pretrained weight matrix. Formally, for a pretrained weight matrix $W \in \mathbb{R}^{k \times n}$, LoRA introduces two low-rank matrices $A \in \mathbb{R}^{k \times r}$ and $B \in \mathbb{R}^{r \times n}$, where LoRA rank $r \ll k, n$, combined as:

$$Y = XW + \alpha SB = XW + \alpha(\widehat{X}A)B \tag{1}$$

where $\widehat{X} \in \mathbb{R}^{m \times k}$ is the input after dropout of $X \in \mathbb{R}^{m \times k}$, $S = \widehat{X}A$ is the intermediate result, m is the number of aggregated tokens (batch size multiplied by sequence length), and α is a constant scalar for scaling. Table 1 summarizes key notations.

When fine-tuning LLMs, LoRA is applied to linear layers, replacing each original layer of dimensions $k \times n$ with a LoRA-equipped version. Assuming half-precision training with full-precision optimizer [73, 81], memory usage of model states per linear layer decreases from $16nk$ bytes to $2nk + 32r(n + k)$ bytes. Since r is much smaller than n and k, the memory footprint of trainable parameters, gradients, and optimizer states is negligible compared to the pre-trained

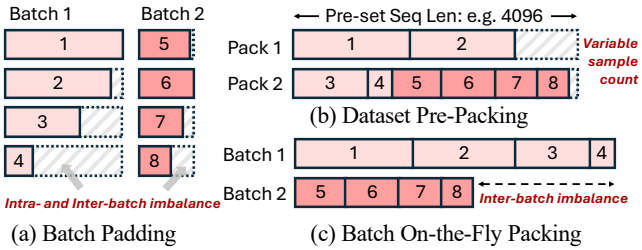
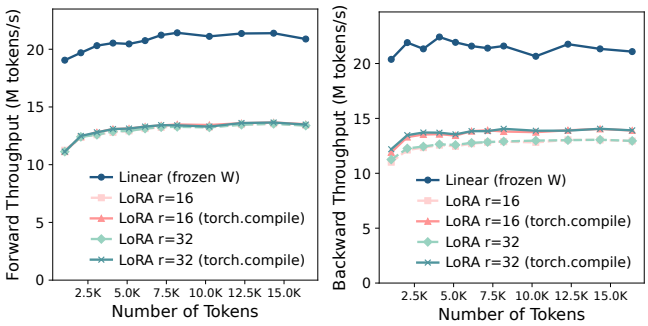


Figure 2. Comparison of (a) traditional batch padding, (b) dataset pre-packing, and (c) batch on-the-fly packing methods for LoRA fine-tuning of LLMs.



(a) Forward pass throughput (b) Backward pass throughput

Figure 3. Throughput comparison of the frozen linear layer (n=k=4096) vs. the corresponding LoRA linear layer with different numbers of tokens and ranks.

weights. For instance, with n=k=4096 and r=16, LoRA parameters account for just about 0.39% of the pre-trained weights. This dramatically reduces the memory required for gradients and optimizer states, decreasing memory demands by nearly 8× compared to full-model fine-tuning.

2.2 System Optimization for Fine-tuning

Despite algorithmic advancements in LoRA [14, 37, 52], and optimizations for LoRA serving [9, 79, 94], existing system optimizations for LoRA fine-tuning primarily reuse techniques originally developed for LLM pre-training, including parallelization [1, 42, 73, 81, 104, 106], kernel fusion [3, 5, 11, 13, 18, 31, 84], and data packing [38, 90].

Parallelization. Parallelization partitions the computation and memory usage of the training process across multiple GPUs. Data parallelism (DP) [1, 42] replicates the model across GPUs and partitions data batches. Fully Sharded Data Parallelism (FSDP or ZeRO-3) [73, 104] partitions model states and communicates them only when necessary, significantly reducing memory usage. Tensor Parallelism (TP) [63, 81] splits linear layers across GPUs and merges partial results via communication. Pipeline Parallelism (PP) [21, 33, 62] divides the model into sequential stages executed in a pipeline manner, reducing communication overhead but potentially introducing idle time (pipeline bubbles).

Kernel Fusion. Kernel fusion improves efficiency by merging multiple operations into fewer GPU kernels, reducing memory transfers and kernel launch overhead [3, 5, 11, 13, 18, 31, 84]. Recent techniques like Flash-Attention [13] and element-wise kernel fusion [31, 84] significantly improve performance by fusing frequently used operations in LLMs. For LoRA, specialized kernels [9, 79] for multi-adapter serving are proposed, but these kernels do not directly improve fine-tuning throughput, as we discuss in Section 3.

On-the-fly Data Packing. When fine-tuning LLMs, training data often consists of token sequences of variable lengths. As depicted in Figure 2(a), traditional padding aligns shorter samples with padding tokens, causing wasted computations. Figure 2(b) shows the dataset pre-packing, which forms fixed-length batches in advance, but introduces variable sample counts per batch, potentially affecting training stability and randomness if not properly handled [38]. In contrast, on-the-fly packing (Figure 2(c)) dynamically concatenates samples within each batch, avoiding wasted computations while maintaining deterministic training samples per batch. Given its effectiveness and wide usage, we adopt on-the-fly packing throughout our work.

3 Motivation

We identify two key limitations in existing LoRA fine-tuning systems: significant runtime overhead from redundant memory access (§3.1), and missed optimization opportunities in multi-job training scenarios (§3.2).

3.1 Significant Runtime Overhead of LoRA Modules

Although LoRA greatly reduces memory usage by introducing only a small number of trainable parameters, it leads to significant runtime overhead in practice. Figure 3 compares the throughput of a frozen linear layer (n=k=4096) against its LoRA-equipped linear layer with different numbers of tokens and ranks. The dark blue lines represent the throughput of the frozen linear layer, and other lines represent the throughput when LoRA modules are applied. We make several key observations: First, the throughput of LoRA linear modules is consistently lower than that of the frozen linear layer, exhibiting a slowdown of approximately 40% and 36% for forward and backward passes, respectively, regardless of the number of tokens. Second, `torch.compile` [3], which provides compiler-based fusion capabilities in PyTorch, provides zero benefits in the forward pass and only negligible improvements in the backward pass. Third, the choice of LoRA rank (r=16 or r=32) also minimally impacts throughput, indicating that the overhead is dominated by inefficient memory access patterns rather than algorithmic cost.

The above profiling results are surprising. In theory, the LoRA adapter should only incur minimal FLOPs and memory accesses overhead since $r \ll n, k$. To investigate the source of the overhead, we perform detailed profiling on both the layer-level and kernel-level. As shown in Figure 4, the total

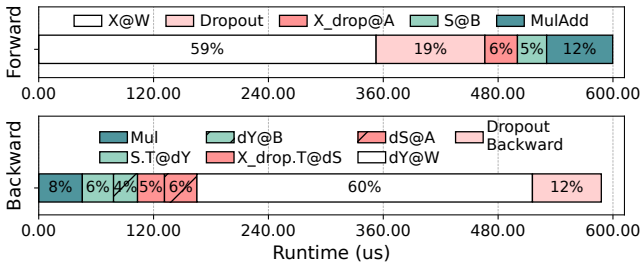


Figure 4. Runtime breakdown of a LoRA linear module with n=k=4096, r=16, and *tokens*=8192. @ is matrix multiplication, d indicates a gradient, and .T represents a transpose.

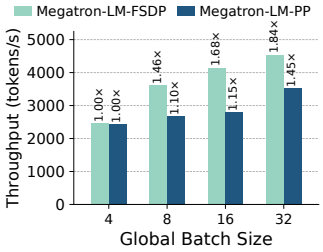


Figure 5. Ideal throughput[1] of LLaMa-3.1-70B on 4 H100 GPUs vs. global batch sizes.

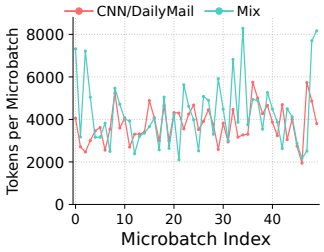


Figure 6. Number of tokens per micro-batch with a fixed micro batch size = 4.

runtime is dominated by three categories of operations: (1) GEMM operations of the frozen linear layer (XW), (2) GEMM operations from LoRA modules (down- and up-projections $\widehat{X}A$ and SB), and (3) other element-wise operations. Although the original GEMM operation still dominates runtime (59% and 60% for forward and backward passes, respectively), the LoRA-specific GEMM operations and element-wise computations introduce substantial overhead. Specifically, LoRA GEMM operations account for 10.76% and 20.37% of overall runtime for forward and backward passes, respectively. These kernels are memory-bandwidth-bound because of the small rank r, and thus the memory read and write of the large activation tensor become the bottleneck. A simple analysis shows that the arithmetic intensity $\mathbb{I}$ of LoRA's down-projection GEMM operation ($\widehat{X}A$) in half-precision is:

$$\text{Arithmetic Intensity } \mathbb{I} = \frac{1}{\frac{1}{r} + \frac{1}{n} + \frac{1}{m}} \ll \mathbb{B} \tag{2}$$

where r, n, and m denote the LoRA rank, output dimension, and batch size, respectively. This intensity $\mathbb{I}$ is far below the machine balance $\mathbb{B}$ (e.g. ~295 for FP16 on NVIDIA H100 GPUs) because of the small r, confirming that performance is bottlenecked by memory bandwidth rather than compute throughput. Moreover, the additional element-wise operations, including dropout, element-wise multiplication, and addition of the partial results from the branches, take 30.46% and 17.49% of the total execution time. These operations are also memory-bound because of the large size of the input and output activation tensors. To further quantify the memory impact, we profile the kernels using NVIDIA Nsight Compute [69], which shows that the total GPU global memory read/write traffic increases by approximately 2.64× compared to the original frozen linear layer.

Therefore, we conclude that the runtime overhead is primarily due to the redundant memory accesses of the large activation tensor relative to their small computational scale. This analysis highlights that while the additional parameters and FLOPs introduced by LoRA seem negligible, the runtime overhead is significant due to the extra memory access.

3.2 Overlooked Opportunities in Multi-LoRA

Multi-LoRA techniques refer to grouping multiple LoRA adapters sharing the same base model from separate tasks into a single batched operation. Such techniques have been widely adopted in LLM serving scenarios to improve GPU utilization by mitigating memory-bandwidth-bound bottlenecks of the frozen GEMM operations caused by small token counts during decoding [9, 79, 94]. However, multi-LoRA techniques are rarely used in fine-tuning. Although mLoRA [98] and Zheng et al. [107] initially used multi-LoRA grouping to reduce the memory footprint of replicated pre-trained models and improve training efficiency, its broader implications for performance optimization remain overlooked. In this section, we first identify two key opportunities where multi-LoRA can significantly improve training efficiency: reducing distributed training overhead and improving GPU load balance. We also explicitly analyze the limitations of existing multi-LoRA techniques.

Mitigating Distributed Parallelism Overhead. By grouping adapters from multiple jobs, multi-LoRA can significantly increase global batch sizes with independent groups of tokens. As Figure 5 illustrates, increasing global batch size from 4 to 32 enhances ideal throughput by 84% and 45% for FSDP and PP, respectively. This improvement occurs because larger batch sizes improve computation-communication overlap in FSDP and reduce pipeline bubbles in PP, significantly reducing the overhead of distributed parallelism. Additionally, since adapters from independent jobs have no interdependencies, multi-LoRA naturally enables near-zero pipeline bubbles by fully utilizing pipeline stages. While mLoRA [98] primarily focused on reducing memory usage on mid-range clusters and implementing uniform adapter filling in pipeline parallelism, our approach broadens these insights to address overhead in general distributed parallelism scenarios.

Reducing Load Imbalance Across GPUs. In ideal scenarios, tokens per micro-batch are uniform, but real workloads often have significant variations, causing load imbalance. Figure 6 shows token counts per micro-batch size of 4 for two datasets: CNN/DailyMail [78], a common summarization dataset, and a Mix combining three summarization datasets

[1] "Ideal" assumes uniform tokens per microbatch and no load imbalance.

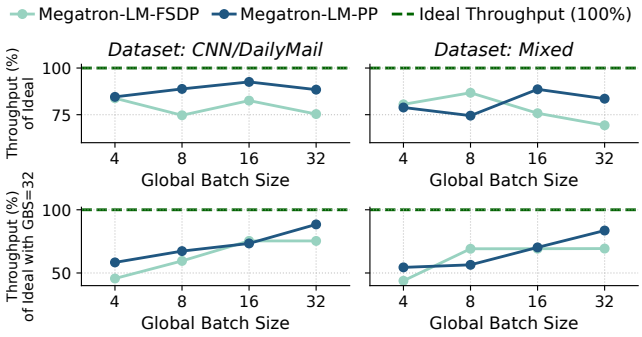


Figure 7. Performance slowdown of practical LoRA fine-tuning of LLaMa-3.1-70B on 4 H100 GPUs compared to ideal fixed-length distributed training scenarios.

(XSum [61], CNN/DailyMail [78], and WikiSum [12]). Detailed length distributions are provided in Figure 13. The substantial variation in tokens per micro-batch creates critical load imbalance in distributed training, limiting performance to the slowest GPU. In FSDP, different ranks process different microbatches but must synchronize before each layer, while in pipeline parallelism, imbalance creates pipeline bubbles across stages. Both scenarios degrade performance. As shown in the top sub-figures of Figure 7, practical LoRA fine-tuning experiences a significant slowdown (up to ~30%) compared to ideal fixed-length distributed training when token counts are imbalanced.

With multi-LoRA fine-tuning, the global batch size includes more samples, creating an opportunity to batch them in a way that balances token counts per microbatch. This mitigates the impact of sequence length variability. Figure 7 (bottom) further illustrates the theoretical ideal throughput improvements achievable (up to 2.28×) by effectively addressing both inefficiencies through multi-LoRA fine-tuning.

Limitations of mLoRA. While mLoRA [98] represents an important step towards multi-LoRA fine-tuning, its design has several limitations that hinder performance and scalability. First, mLoRA assumes uniform adapter grouping and schedules according to memory capacity, but does not handle load imbalance from variable sequence lengths found in real workloads. Furthermore, its `BatchLoRA` kernel reduces kernel launch overhead but still does not solve the core memory redundant access bottleneck identified in Section 3.1. Finally, mLoRA's design is narrowly focused on Pipeline Parallelism and relies on inefficient CPU-based communication, which limits scalability on modern GPU clusters with high-bandwidth interconnects like NVLink [67].

4 Overview and Key Ideas

To address the two key inefficiencies identified in Section 3: runtime overhead from redundant memory access and throughput degradation due to load imbalance and parallelism overhead, we propose *LoRAFusion*, a novel LLM LoRA fine-tuning system that improves system throughput via kernel-level

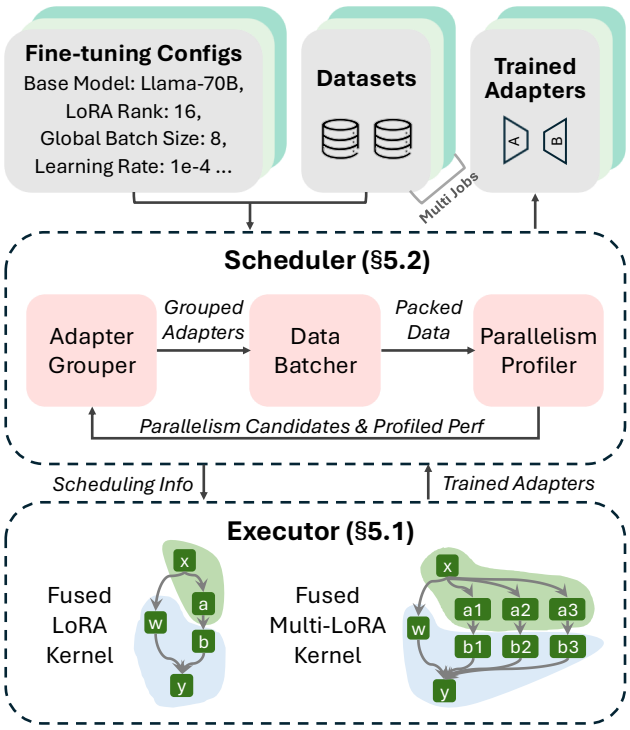


Figure 8. Overview of LoRAFusion.

optimizations and job-level scheduling. In this section, we summarize the two key ideas behind it:

1. FusedLoRA and FusedMultiLoRA. To reduce LoRA's runtime overhead, LoRAFusion fuses memory-bound operations while preserving performance for compute-bound matrix multiplications by carefully splitting the computation graph. This is motivated by the key bottleneck in LoRA, i.e., redundant memory accesses to large activation tensors. A naive solution is to fuse the entire computation graph into a single kernel. However, doing so introduces costly recomputation or synchronization overhead [80]. It also consumes GPU resources like registers and shared memory, which degrades GEMM performance due to suboptimal tiling.

LoRAFusion addresses this by introducing a graph-splitting strategy. Instead of full fusion, it splits the graph at the intermediate tensors with LoRA rank r, which are small and cheap to materialize. This enables fusion around full-sized activations without recomputation or synchronization. Thus, the resulting FusedLoRA kernels reduce memory traffic while preserving the performance of frozen GEMMs.

We further extend this idea to support multiple LoRA adapters using a tile-level routing mechanism. The proposed FusedMultiLoRA processes tokens from different jobs within the same fused kernel. Each tile uses a precomputed mapping to dynamically select the appropriate adapter weights. This avoids the need for separate kernel launches per adapter and maintains high GPU utilization across jobs. During backpropagation, gradients are routed similarly. The FusedMultiLoRA forms the foundation of LoRAFusion's job-level scheduling.

2. Multi-LoRA Scheduling. To exploit the benefits of multi-LoRA fine-tuning, LoRAFusion introduces a scheduler that coordinates adapter grouping and data batching. The major challenge is to construct well-balanced microbatches across jobs while respecting data dependencies between global batches. Specifically, in pipeline parallelism, each global batch must wait for all samples from the previous one to complete their backward passes. LoRAFusion addresses this with a two-stage hierarchical strategy: it first groups adapters in a way that keeps their batches spread apart in the schedule, then solves a bin-packing problem to batch their samples and reduce imbalance.

In the first stage, LoRAFusion groups LoRA adapters based on their sample length distributions to ensure that consecutive global batches from the same adapter are sufficiently spaced in the schedule. In the second stage, samples within each group are packed into microbatches using a two-step MILP-based optimization, with the greedy algorithm as a fallback and multiprocessing for efficiency. A final merge pass reduces underfilled microbatches when possible. This scheduling strategy improves load balance and reduces pipeline stalls, boosting overall system throughput.

System Workflow. Figure 8 presents LoRAFusion's system workflow. Given a set of fine-tuning jobs, LoRAFusion first extracts dataset statistics and proposes a microbatch token budget via a parallelism simulator. It then forms adapter groups and constructs microbatches accordingly. The grouping and batching outputs are re-evaluated through simulation, and the process iterates until a high-throughput configuration is found. Finally, jobs are executed using the fused kernels described above. A multi-adapter runtime coordinator ensures token-to-adapter consistency, manages resource sharing, and tracks gradients across job boundaries. Through the combination of fused execution and coordinated job scheduling, LoRAFusion addresses both the memory and parallelism bottlenecks in LoRA fine-tuning, improving throughput while maintaining correctness and generality.

5 System Design

5.1 FusedLoRA and FusedMultiLoRA

As described in Section 3.1, LoRA modules introduce significant runtime overhead despite adding only a small number of parameters. Our profiling reveals that this overhead stems primarily from redundant memory access. Our goal is to fuse operations within the LoRA computation graph to reduce these memory transfers while preserving compute efficiency.

Design Considerations and Challenges.

While kernel fusion can reduce redundant memory access, fusing all LoRA operations into a single kernel introduces practical challenges. First, the frozen GEMM operation $Y_1 = XW$ is compute-bound and highly sensitive to kernel tiling strategies and GPU resource usage (e.g., shared memory and register file). A suboptimal tiling layout or overuse of registers and shared memory can greatly degrade its performance. Second, fusing operations with producer-consumer dependencies, such as $\widehat{X}A$ followed by $(\widehat{X}A)B$, may require recomputing intermediate results or introducing thread block synchronization, both of which can add overhead [80]. Thus, a good fusion strategy must minimize memory access while preserving optimal compute performance and avoiding expensive synchronization or recomputation.

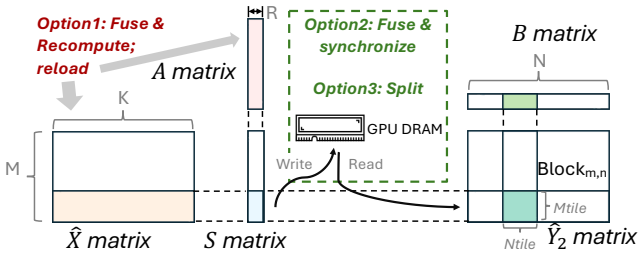
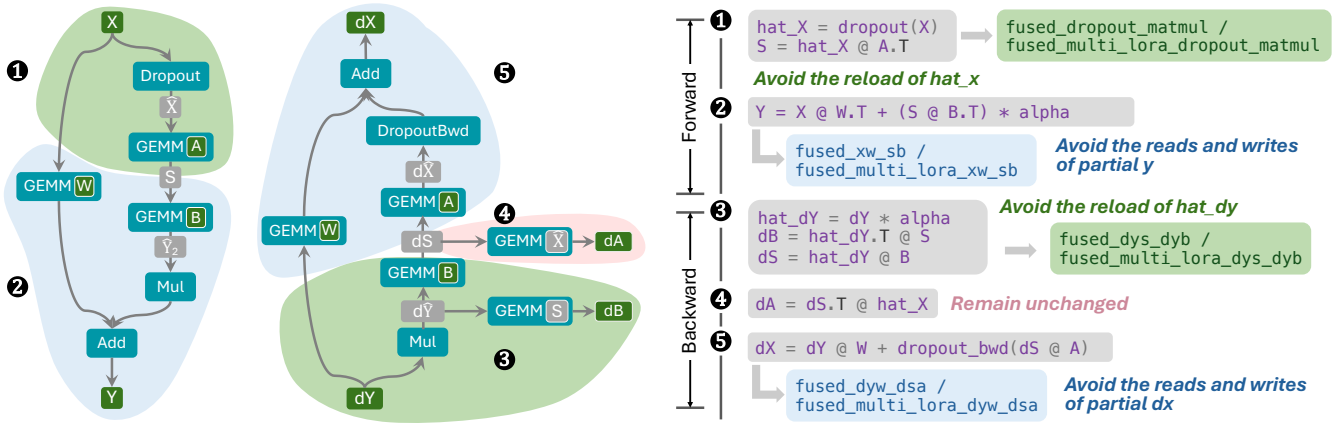


Figure 9. Overview of our fusion strategy for LoRA modules in the forward pass, illustrating the full graph fusion approach vs. the split graph fusion approach.

Full Graph Fusion vs. Split Graph Fusion. Figure 9 illustrates three design choices for handling the intermediate tensor $S = \hat{X}A$ in the forward pass. The first option recomputes S inside each tile of the fused kernel, but requires loading the entire A matrix repeatedly, becoming expensive when batch size M is large. The second option fuses computation and uses synchronization across thread blocks to share S, where only a single $Mtile$ (the m-th block in the token dimension) computes the intermediate S tiles and writes to global memory, while other tiles wait through a semaphore. This adds coordination overhead. Our approach takes a third option: explicitly storing and reloading S from GPU global memory. Since S is much smaller than other tensors and depends on the small LoRA rank r, the cost of reading and writing it is low. Splitting the graph at S avoids both recomputation and synchronization while reducing expensive memory traffic associated with full-sized activation tensors. This approach preserves GPU resources for optimal tiling of the compute-bound XW operation, maintaining peak performance for the most computationally intensive part.

FusedLoRA Design. Figure 10 illustrates our fused kernel design for both forward and backward passes. In the forward pass (Figure 10(a)), we combine dropout and down-projection into a single kernel (❶) to eliminate reloading of the full-sized activation tensor. We also fuse the compute-bound base model GEMM ($Y_1 = XW$) with the memory-bound LoRA operations ($Y_2 = \alpha SB$) (❷). This fusion eliminates redundant memory operations and saves one read and write of the full-sized output tensor by directly accumulating partial results, without affecting the base GEMM performance. In the backward pass, we apply similar principles. Operation ❸ fuses the gradient computation of S and B, eliminating



(a) Forward Graph (b) Backward Graph (c) Optimization Opportunities and Fused Kernel Strategies

Figure 10. LoRA kernel design. FusedLoRA reduces memory accesses by combining memory-heavy LoRA branches with base GEMM operations on frozen weights. The right figure has transposed weight tensors to match the hardware memory layout.

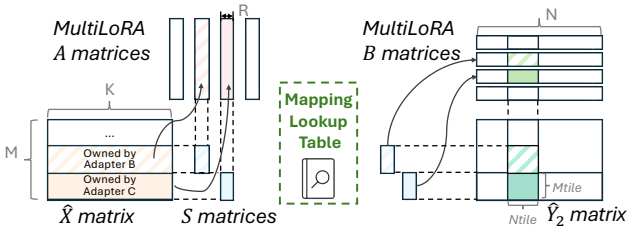


Figure 11. Illustration of FusedMultiLoRA in the forward pass. The routing of LoRA adapters is done at the tile level.

the need to reload dY. Operation ❹ remains separate since it operates on the small masked input, where fusion provides minimal benefit. Operation ❺ horizontally fuses the compute-intensive gradient computation for the base model with memory-bound LoRA path operations, preventing redundant reads and writes of partial output gradients. The key insight of our approach is strategically identifying operations where horizontal fusion reduces memory traffic without compromising computational efficiency. By fusing operations that share large tensors (❷, ❸, and ❺), we significantly reduce memory bottlenecks while maintaining optimal tiling strategies for compute-bound operations.

Extending to FusedMultiLoRA. To support concurrent fine-tuning of multiple LoRA adapters, we extend FusedLoRA to FusedMultiLoRA, allowing our fused kernels to operate on mixed-adapter batches from different jobs. As shown in Figure 11, each input $Mtile$ is tagged with an adapter ID and configuration, such as LoRA rank, scaling factor, and dropout ratio, stored in a lightweight lookup table. During execution, the frozen model computation is shared across all tokens, while adapter-specific logic is applied dynamically per $Mtile$. For each $(Mtile, Ntile)$ tile of the output, the kernel loads the appropriate A and B matrices and applies the correct scaling and dropout. In the backward pass, the same mapping is used to route gradients to their respective adapters without interference. This tile-level routing allows efficient execution of heterogeneous adapters in a single fused run, avoiding redundant computation and enabling the job-level optimizations introduced in Section 5.2.

FusedLoRA reduces memory traffic by fusing LoRA operations around shared activations while preserving base model efficiency. Building on FusedLoRA, FusedMultiLoRA supports heterogeneous adapters via tile-level routing. The system dynamically chooses between them, falling back to FusedLoRA when only one adapter is present in the batch.

5.2 Multi-LoRA Scheduler

LoRAFusion not only reduces runtime overhead through fused kernels but also improves end-to-end throughput by scheduling multiple LoRA fine-tuning jobs together. This is achieved by grouping adapters and adaptively batching their samples to balance GPU load and minimize distributed parallelism overhead. Figure 12 shows the overall process. Adapters are grouped based on sequence length statistics (top), and adapters in the same group are trained jointly. For each group, we aggregate samples into global batches and pack each one into microbatches using a two-stage MILP-based optimization (middle). Once microbatches are generated in parallel with multiprocessing, a final merge pass combines underfilled microbatches across global batches when data dependencies allow (bottom).

Granularity. Due to data dependencies between consecutive global batches, our scheduling operates at the granularity of individual global batches. Each adapter's dataset is divided into global batches based on the user-specified global batch size. We then aggregate all samples belonging to the same global batch index across adapters and pack them into multiple microbatches.

Bubble Lemma & Adapter Grouping. LoRAFusion first groups LoRA adapters before batching samples to reduce scheduling complexity. In pipeline parallelism with S stages,

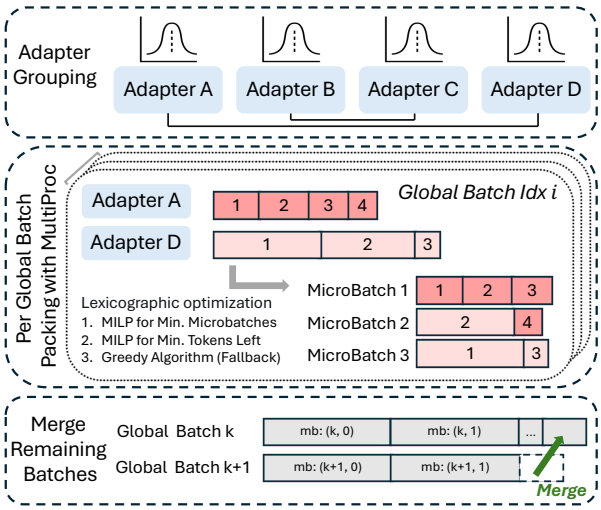


Figure 12. Multi-LoRA adapter scheduling workflow. Top: Adapter grouping by sequence length statistics. Middle: Two-stage MILP optimization for microbatch creation. Bottom: Cross-batch merging of underfilled microbatches.

a sample's backward pass begins only after $S-1$ other microbatches complete forward passes. To maintain data dependencies, we define the *bubble lemma*: for adapter i, if sample s from global batch j is committed at microbatch k, no sample from batch j+1 of the same adapter can be committed before microbatch k+S-1 (after sample s's backward pass completes). Without constraints, adaptive batching might scatter samples from consecutive batches across microbatches. The tail of batch j could conflict with batch j+1's head, causing incorrect execution. We resolve this conflict by grouping adapters with strict ordering between groups, while allowing flexible merging within each group. This ensures batches from the same adapter are separated by microbatches from other groups, satisfying the bubble condition. For load balance within groups, we use head-tail pairing, sorting adapters by mean token length, and pairing short-sequence adapters with long ones. This grouping balances constraints with flexibility and forms our data packing foundation.

Data Batching with Two-Stage MILP. After adapters are grouped, we solve a bin-packing problem to batch samples into microbatches, each constrained by a fixed token capacity. Our goal is to reduce both the total number of microbatches and the impact of underfilled ones, which affect load balance and pipeline utilization. Specifically, we aim to (i) minimize the number of microbatches needed to pack all samples, and (ii) make the smallest microbatch as empty as possible to enable better merging in later stages.

We address the bin-packing problem using a two-stage mixed-integer linear programming (MILP) formulation (see Algorithm 1, lines 3-7). For notational brevity, let P denote the padding multiple, which is a user-specified parameter to pad the sequence length belonging to the same adapter to a multiple of P (e.g., 64 or 128). Let $x_{s,b} \in \{0,1\}$ denote

Algorithm 1: Data Batching & Merging (Per Group)

Data: Adapters with grouped samples, token capacity C, timeout t
Result: Scheduled microbatches satisfying pipeline constraints

1 **foreach** *global batch b in parallel* **do**
2 *// Greedy fallback as baseline*
2 $(B_g, \{m_i^g\}) \leftarrow$ `GreedyPacking`(b, C)
3 *// Stage 1: minimize number of microbatches*
3 $B^* \leftarrow$ `MILP_MinBins`$(b, C, \text{timeout} = t)$
4 **if** $B^* \geq B_g$ **then**
5 $B^* \leftarrow B_g$
6 **end**
7 *// Stage 2: minimize smallest bin tokens*
7 $\{m_i\}_{i=1}^{B^*} \leftarrow$ `MILP_MinSmallestBin`$(b, B^*, C, \text{timeout} = t)$
8 **if** $B^* = B_g$ ***and*** $\min_i m_i \geq \min_i m_i^g$ **then**
9 **return** `GreedyPacking`(b, C)
10 **end**
11 **end**
12 **foreach** *consecutive batch pairs* $(b, b{+}1)$ **do**
13 Shift tokens from b+1 into b if bubble lemma is preserved
14 **end**
15 `VerifyAndFix(schedule)` *// Insert no-ops where needed*
16 **return** Scheduled microbatches

whether sample s is assigned to bin b, $k_{a,b} \in \mathbb{N}$ be the padded multiples contributed by adapter a in bin b, and $z_b \in \{0,1\}$ indicate whether bin b is used. In the first stage, we solve:

$$\begin{aligned} \arg\min_{x_{s,b},k_{a,b},z_b} \quad & \sum_{b=1}^{B} z_b \\ \text{s.t.} \quad & z_{b+1} \leq z_b \quad \forall b < B \\ & \sum_{b=1}^{B} x_{s,b} = 1 \quad \forall s \in \text{samples} \\ & \sum_{s \in \text{adapter}(a)} \text{len}(s) \cdot x_{s,b} \leq k_{a,b} \cdot P \quad \forall a, b \\ & z_b \leq \sum_{a} k_{a,b} \cdot P \leq \text{capacity} \cdot z_b \quad \forall b \end{aligned} \tag{3}$$

The constraints ensure used bins are contiguous from the start, each sample is assigned to exactly one bin, adapter-specific token counts respect padding multiples, and bin capacity is not exceeded.

With the optimal number of bins B^* from the first stage, the second stage fixes $B = B^*$ and minimizes the smallest total token count among all bins. The second stage solves:

$$\begin{aligned} \arg\min_{x_{s,b},k_{a,b}} \quad & \min_{b \in [1,B^*]} \sum_{a} k_{a,b} \cdot P \\ \text{s.t.} \quad & \sum_{b=1}^{B^*} x_{s,b} = 1 \quad \forall s \in \text{samples} \\ & \sum_{s \in \text{adapter}(a)} \text{len}(s) \cdot x_{s,b} \leq k_{a,b} \cdot P \quad \forall a, b \\ & \sum_{a} k_{a,b} \cdot P \leq \text{capacity} \quad \forall b \end{aligned} \tag{4}$$

This optimization problem can be reformulated into an MILP problem. It minimizes the smallest bin size, which leaves more space in the least-full microbatch for potential merging in later stages, thereby reducing pipeline stalls.

To improve runtime efficiency, we implement two techniques (also reflected in Algorithm 1). First, we set a timeout on the MILP solver and fall back to a greedy bin-packing algorithm if the solver takes too long (lines 2, 5, and 9). Second, since global batches are independent, we parallelize the bin-packing optimization across batches using multiprocessing (line 1), which allows us to efficiently schedule all training data while balancing load and reducing microbatch count.

Merging & Verification. After microbatch packing, the final microbatch in a global batch may be underfilled, reducing GPU efficiency and increasing pipeline bubbles. To mitigate this issue, we apply a greedy merge pass that shifts tokens from the next global batch into the current batch's final microbatch (as shown in Figure 12 bottom), as long as capacity and the bubble lemma are satisfied (Algorithm 1, lines 12-14). We then perform a verification step to ensure no constraint is violated. If any bubble condition is unmet, we insert no-op microbatches to restore correctness and preserve pipeline consistency (line 15).

Parallelism Profiler. The scheduler requires token capacity as input, which depends on the parallelism strategy. LoRAFusion assumes that effective scheduling keeps tokens within each microbatch close to the token capacity. Since token capacity and parallelism strategies are orthogonal to scheduling, they should be tuned outside the scheduler using automatic parallelization techniques [51, 53, 106, 112]. We implement a lightweight profiler that directly benchmarks runtime under different model parallelism configurations with fixed-length inputs and collects throughput. We choose the best-performing configuration and pass its token capacity to the data batching stage, ensuring that microbatch packing aligns with the system's performance characteristics.

6 Evaluation

We implement LoRAFusion with ~10K LoC in Python. The FusedLoRA and FusedMultiLoRA kernels are developed using Triton [84], and multi-adapter pipeline parallelism is built on top of Megatron-LM [81]. Since Megatron-LM does not natively support LoRA, we integrate Hugging Face Transformers [93] and the PEFT Library [55] for model architecture and LoRA adaptation.

The optimizations in LoRAFusion are designed to be lossless, guaranteeing they do not affect model convergence or final quality. Our FusedLoRA and FusedMultiLoRA kernels are numerically stable, producing outputs that are functionally identical to the baseline implementations within numerical precision. While our adaptive scheduler rearranges samples to form balanced microbatches, it strictly preserves the order of global batches, ensuring the sequence of gradient updates remains unchanged. Given that model behavior is identical by design, our evaluation focuses exclusively on system performance metrics like throughput.

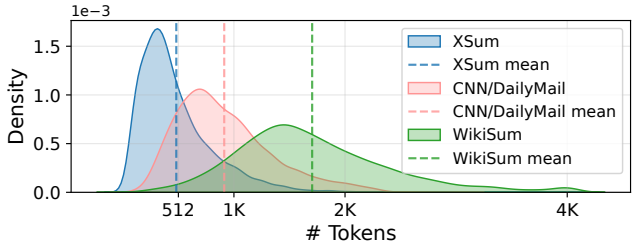


Figure 13. Distribution of sample lengths across the XSum [61], CNN/DailyMail [78], and WikiSum [12] datasets used for LoRA fine-tuning.

We evaluate LoRAFusion across a range of real-world fine-tuning scenarios involving multiple datasets, model scales, GPU platforms, and job configurations. Our primary metric is throughput, measured in trained tokens per second, which better reflects system efficiency for inputs with sequence length variations. Finally, we perform detailed scalability studies, ablation studies, and performance breakdowns to analyze the contribution of each system component.

6.1 Methodology

Hardware Settings. We primarily benchmark LoRAFusion on a GPU cluster with NVIDIA H100 (80GB) GPUs and additionally report results on L40S (48GB) GPUs to demonstrate generalizability. Each H100 node is equipped with 8×H100 GPUs connected via NVLink, 208 vCPUs, and Infiniband for multi-node communication. Each L40S server contains 4×L40S GPUs connected over PCIe and 128 vCPUs. Most experiments use the smallest number of GPUs that fit the model and maintain good utilization, typically 1, 2, or 4 GPUs. As shown in our Scalability Studies (Section 6.3), assigning fewer GPUs per job and using additional GPUs to run more independent jobs often leads to better efficiency, as it reduces inter-GPU communication and synchronization overhead. The Scalability Studies also show LoRAFusion is fully compatible with both data-parallel and multi-node scaling.

Workload Settings. We evaluate LoRAFusion on three open-source language models of varying sizes: LLaMa-3.1-8B [56], Qwen-2.5-32B [97], and LLaMa-3.1-70B [56]. All experiments use summarization as a representative sequence-to-sequence task, which is widely used in prior LoRA fine-tuning studies [32, 45, 50]. We select three public summarization datasets: XSum [61], CNN/DailyMail [78], and WikiSum [12]. These datasets have diverse length distributions, as shown in Figure 13, which stresses batching and scheduling under realistic conditions. For multi-LoRA experiments, we train four LoRA adapters in parallel. In the XSum, CNN/DailyMail (CNNDM), and WikiSum configurations, all four adapters are trained independently on the same dataset. In

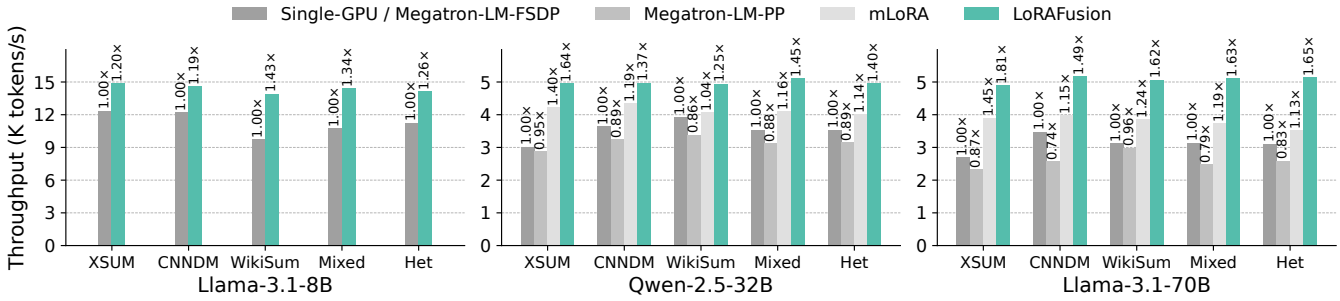


Figure 14. End-to-end training throughput (tokens/sec) of training 4 LoRA adapters on 1, 2, and 4 H100 GPUs. The first four bars per subfigure represent homogeneous workloads (same dataset), and the final (Het) shows heterogeneous adapters trained on different datasets.

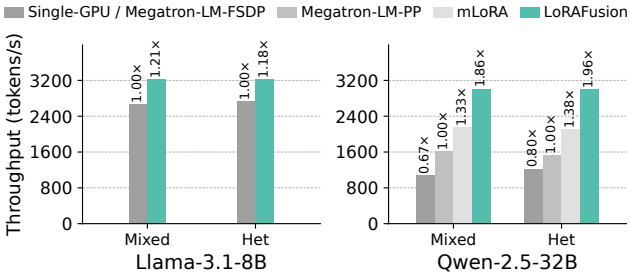


Figure 15. End-to-end training throughput (tokens/sec) of training 4 LoRA adapters on 1 and 4 L40S GPUs.

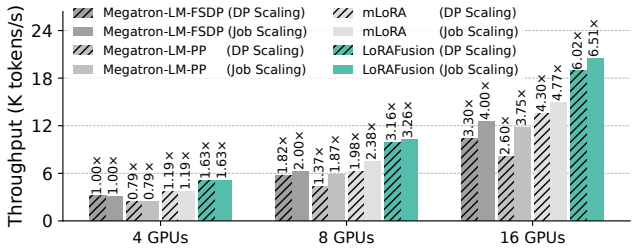


Figure 16. Scalability of LoRAFusion across 4, 8, and 16 H100 GPUs when training 4 LoRA adapters simultaneously. DP scaling means the more GPUs are used to increase the DP degree for the same job, while Job scaling means different LoRA fine-tuning jobs are scheduled to utilize more GPUs. Global batch sizes are scaled proportionally with GPU count to ensure fair comparison.

the Mixed setting, each adapter is trained on a dataset combining samples from all three. In the Heterogeneous (Het) setting, the four adapters are trained on different datasets: one each on XSum, CNN/DailyMail, WikiSum, and Mixed.

Baselines. We compare LoRAFusion against three baselines: (i) Megatron-LM [81] with fully sharded data parallel (FSDP), (ii) Megatron-LM with pipeline parallelism (PP), and (iii) mLoRA [98]. Megatron-LM does not support multi-LoRA fine-tuning natively, so tasks are trained sequentially, while mLoRA supports multi-LoRA fine-tuning. The original mLoRA uses Python RPC for inter-GPU communication, which performs poorly on NVLink-equipped GPUs. Therefore, we reimplement mLoRA inside our system with high-performance communication primitives to ensure fair comparison. In addition, since mLoRA does not provide a unique multi-LoRA CUDA kernel, we optimistically assume it has the same performance as the naive single LoRA kernel. Tensor parallelism is not evaluated due to the lack of efficient support in existing LoRA frameworks. All experiments use PyTorch 2.6, CUDA Toolkit 12.4, Triton 3.2.0, and Megatron-Core 0.11.0.

6.2 End-to-End Results

Speedup on H100 GPUs. Figure 14 reports the end-to-end throughput of training 4 LoRA adapters across three models. LoRAFusion consistently outperforms all baselines by $1.19 - 1.96\times$. For LLaMa-3.1-8B, which fits on a single H100 GPU, LoRAFusion achieves an average $1.26\times$ speedup (up to $1.43\times$), primarily from the FusedLoRA kernel, which reduces memory traffic. Since single-GPU setups do not suffer from load imbalance, the improvement here directly reflects kernel-level gains. LoRAFusion achieves high speedup on the WikiSum dataset due to the large variance in sample lengths. While the baseline methods suffer from out-of-memory errors, LoRAFusion achieves stable packing. For Qwen-2.5-32B and LLaMa-3.1-70B, which require distributed training, LoRAFusion achieves $1.42\times$ and $1.64\times$ average speedup (up to $1.64\times$ and $1.81\times$) respectively. Larger models benefit more from improved scheduling, as pipeline stalls and load imbalance become more pronounced at higher parallelism. In the most challenging heterogeneous setting (Het), where each adapter uses a different dataset, LoRAFusion still achieves strong performance, highlighting its robustness.

Speedup on L40S GPUs. Figure 15 presents results on NVIDIA L40S GPUs. LoRAFusion achieves $1.19 - 1.91\times$ average speedup for LLaMa-3.1-8B and Qwen-2.5-32B respectively. The benefit is smaller for LLaMa-3.1-8B due to limited memory capacity on a single L40S GPU, which constrains batch size and limits kernel fusion effectiveness. However, even under such constraints, LoRAFusion maintains consistent improvements, demonstrating generalizability across model sizes and hardware platforms.

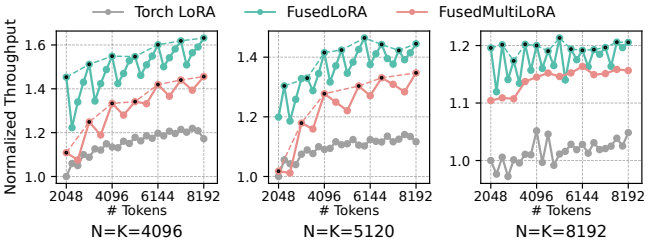


Figure 17. Performance of FusedLoRA kernel in forward and backward passes.

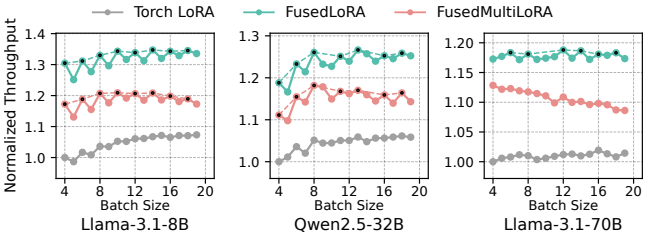


Figure 18. Performance of FusedLoRA kernel in decoder layers of different models.

6.3 Scalability Studies

We evaluate LoRAFusion on 4, 8, and 16 H100 GPUs under two scaling strategies: DP scaling (more GPUs per job) and job-level scaling (more concurrent jobs). 16 H100 GPUs experiment is conducted on 2 nodes each with 8 GPUs, connected via InfiniBand [66]. Global batch size is scaled with GPU count in both settings. We draw two key conclusions. First, job-level scaling consistently outperforms DP scaling due to better load balance, achieving 1.18× and 1.25× higher throughput on 8 and 16 GPUs, respectively. Second, LoRAFusion is fully compatible with DP scaling and multi-node fine-tuning, and still delivers strong performance, achieving 1.78× average speedup over Megatron-LM and 1.50× over mLoRA under DP scaling.

6.4 Effectiveness of FusedLoRA Kernel

Kernel Performance. Figure 17 shows the throughput of our FusedLoRA and FusedMultiLoRA kernels compared to the standard Torch LoRA implementation [55]. FusedLoRA achieves an average speedup of 1.27× (up to 1.39×), while FusedMultiLoRA achieves 1.17× on average (up to 1.24×). In the forward pass, FusedMultiLoRA performs similarly to FusedLoRA, as most computation is shared. In the backward pass, it incurs slight overhead from accumulating gradients across adapters and additional element-wise operations. Despite this overhead, both kernels consistently outperform the baseline across different token sizes and model configurations.

Layer-wise Performance. Figure 18 compares the speedup across different linear layers in the model. FusedLoRA achieves an average speedup of 1.21× (up to 1.30×), while FusedMultiLoRA achieves 1.13× (up to 1.17×). These results

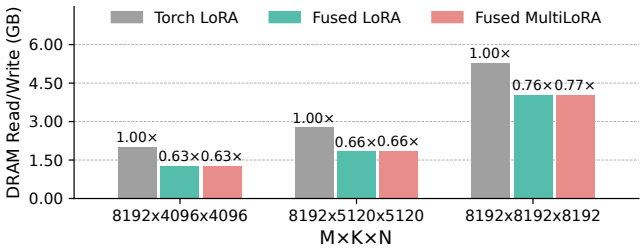


Figure 19. GPU DRAM memory traffic comparison between different kernels from NVIDIA Nsight Compute (NCU).

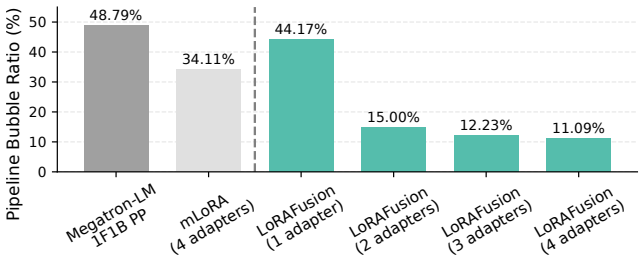


Figure 20. Pipeline bubble ratio under different methods.

are based on microbatches containing four adapters. In practical fine-tuning workloads, each microbatch typically contains only one or two adapters, making FusedMultiLoRA's performance close to FusedLoRA.

Memory Traffic Reduction. Figure 19 shows DRAM read and write traffic from NVIDIA Nsight Compute (NCU) across representative GEMM shapes. Both FusedLoRA and FusedMultiLoRA consistently reduce memory usage compared to Torch LoRA. For example, on the 8192×4096×4096 shape, total DRAM traffic reduces to 0.63×. Across all settings, traffic is reduced by 34% – 37%, confirming that our fusion design effectively reduces redundant memory access.

Performance Insights Across Diverse Hardware. The FusedLoRA and FusedMultiLoRA kernels reduce redundant memory access for large activation tensors, which is especially important on hardware where memory bandwidth is much lower compared to compute FLOPS. As modern accelerators increase compute FLOPS faster than memory bandwidth [27], the benefits of our fused kernels are expected to grow in future systems.

6.5 Effectiveness of Job-Level Scheduling

Pipeline Bubble Reduction. Figure 20 shows how LoRAFusion helps reduce pipeline bubbles by scheduling multiple adapters together. We make three key observations. First, with only one adapter, the bubble ratio remains high at 44.17%, close to Megatron-LM's 48.79%. This is because grouping is ineffective when only one dataset is available, showing the importance of multi-LoRA for improved scheduling flexibility. Second, as more adapters are trained together, the bubble ratio steadily decreases: 15.00% for 2 adapters, 12.23% for 3, and 11.09% for 4. In comparison, mLoRA reaches only 34.11%, confirming that LoRAFusion's

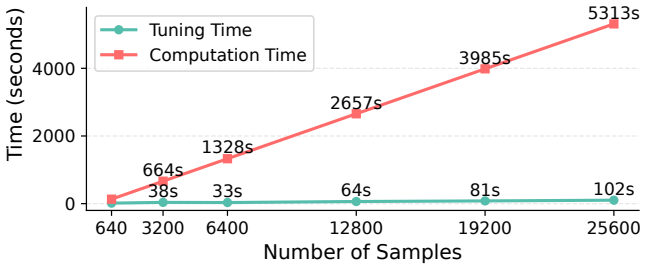


Figure 21. Tuning and computation time vs. number of samples for 4-stage pipeline with 4 adapters.

grouping and batching significantly reduce pipeline idle time. Lastly, with four adapters, the bubble ratio is 11.09%. This is due to uneven execution times across pipeline stages, with the last stage taking longer because it handles an extra linear layer and cross-entropy loss. This limitation is not solvable by our scheduler and is out of scope for this work.

Tuning Time. Figure 21 shows how tuning and computation time grow with the number of training samples for a 4-stage pipeline with 4 adapters, measured on 64 vCPUs and 4 H100 GPUs. The scheduling time increases nearly linearly, from 15.74 seconds at 640 samples to 102.12 seconds at 25600 samples, demonstrating linear scalability of our scheduler. The computation time also increases nearly linearly, with a much larger slope than the scheduling time. The scheduling overhead is negligible for three reasons. First, the CPU-based scheduling runs in parallel with GPU training of the preceding global batch, with linear scaling of 4ms per sample in CPU and magnitude difference in execution time between CPU and GPU, making the scheduler's latency fully hidden by this overlap. Second, as shown in Figure 20, the performance gains saturate at 4 adapters, allowing practical deployment with a small constant number of adapters. Third, we implement a timeout on the MILP solver and fall back to a greedy bin-packing algorithm if the solver takes too long, allowing us to configure the scheduler to balance effectiveness and efficiency, ensuring the scheduling overhead is always within a controllable range.

Effectiveness of the Merging & Greedy Fallback. We evaluate the effectiveness of our scheduler's merging and greedy fallback components on 4 adapters of LLaMa-3.1-70B fine-tuned on four H100 GPUs. The merging pass improves throughput by 4.34%, while the two-stage MILP optimization provides an additional 3.82% improvement over pure greedy bin-packing. The MILP solver path is selected for 77.4% of global batches with a timeout of 10 seconds, indicating its effectiveness in reducing token counts for underfilled microbatches. These modest improvements reflect that most microbatches are already well-packed, and our algorithms primarily optimize the final microbatch in each global batch. Since scheduling overhead is hidden by parallel GPU execution, these optimizations push performance toward the hardware limit without introducing additional overhead.

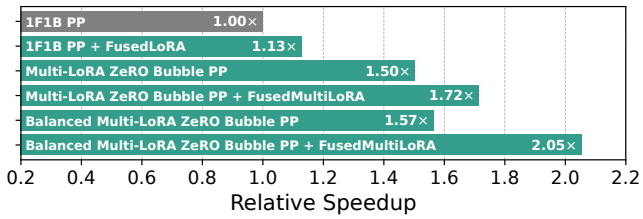


Figure 22. Speedup breakdown of LoRAFusion on LLaMa-3.1-70B with 4 GPUs.

6.6 Speedup Breakdown

Figure 22 shows the contribution of each component in LoRAFusion. Starting from the baseline one forward one backward (1F1B) pipeline parallelism used in Megatron-LM, adding FusedLoRA alone yields a 1.13× speedup. This modest gain is constrained by load imbalance and suboptimal token shapes, which limit kernel efficiency (see Figure 17). Replacing 1F1B with Multi-LoRA zero-bubble pipeline parallelism improves throughput to 1.50× by eliminating pipeline stalls through more microbatches from independent adapters. Adding FusedMultiLoRA kernel further raises the speedup to 1.72× by enabling multi-adapter microbatches and reducing redundant memory access. When we apply our scheduler to rebalance token distribution across microbatches, performance improves to 1.57× even without fusion, as it significantly reduces load imbalance. Finally, combining adaptive scheduling with fused kernels achieves the highest speedup of 2.05×, showing the importance of jointly optimizing kernel efficiency, parallelism, and workload balance.

The speedup over mLoRA is driven by two main optimizations: (i) our kernel fusion yields a 1.15× speedup, as seen by comparing bars 3 and 4 in Figure 22, with even greater gains when the sequence length is regular and matches the performant sequence length of our kernel (bars 5 and 6); and (ii) our adaptive batching mitigates load imbalance, providing a 1.19× speedup (bars 4 and 6). These improvements are further supported by microbenchmarks in Figure 17 (1.17× average speedup, up to 1.24× for kernel performance) and Figure 20 (23.02% reduction in pipeline bubbles).

7 Discussion and Future Work

Generalizability to LoRA Variants. Our kernel fusion design is extensible to other popular LoRA variants like DoRA [52] and VeRA [37]. These methods typically add pre- or post-processing functions around the core LoRA computation. Our optimizations are orthogonal to these modifications, and users can define prologue/epilogue functions to extend our kernels. While manual extension is effective, a more general approach is to integrate our fusion patterns into a compiler framework. As future work, we plan to leverage `torch.compile` by adding compiler annotations as hints that guide the fusion process of the LoRA pattern. This would automate the optimization for both existing and future LoRA

variants, eliminating the need for manual kernel development and system expertise from users.

Generalizability to Quantization. The kernels proposed in LoRAFusion can be directly applied to 4-bit QLoRA [14]. Current QLoRA implementations dequantize 4-bit weights to half-precision before LoRA computation, allowing our kernels to work without modification. While dequantization could be fused with the LoRA path, recent work shows that two-step approaches are often more performant for large token counts [17].

8 Related Work

System-level Optimizations on LoRA. While there has been extensive work at the algorithmic level to make fine-tuning more stable and efficient, system-level optimizations for LoRA are mostly for inference and serving. PetS [110] is the first work that proposes multi-task parameter-efficient fine-tuned transformers serving and introduces a scheduling algorithm to coordinate different requests. Punica [9], S-LoRA [79], and dLoRA [94] propose serving multiple LoRA adapters together to increase system throughput. For LoRA fine-tuning, research interest is more focused on privacy preserving. Offsite-tuning [96] and DLoRA [26] propose to decouple the large model owner and the data owner and connect them with lightweight adapters to enhance privacy. In concurrent work, LobRA [47], explored addressing multi-tenant fine-tuning over heterogeneous data. Its scheduling approach is complementary to our contributions. Our kernel fusion provides an orthogonal optimization that can directly enhance LobRA's performance, while our pipeline-aware scheduler could further reduce pipeline bubbles within LobRA as an additional improvement.

Model Batching in Training. Model batching improves hardware utilization by co-scheduling multiple training jobs on shared hardware. AutoML frameworks [23, 24] and TUPAQ [82] train numerous candidate models in parallel for architecture search, while Ease.ml [43] focuses on multi-tenant model-selection. HFTA [91] proposes to horizontally fuse models from repetitive jobs at the operator level for better hardware utilization. Multi-tenancy has also been applied to federated learning [113]. Unlike these general approaches, which are not model-aware, LoRAFusion is tailored for multi-tenant LoRA fine-tuning. It exploits the shared base model to reduce memory usage and address distributed training bottlenecks like pipeline bubbles and load imbalance.

Kernel Fusion. Kernel fusion is widely used to reduce redundant memory access and improve performance. Compiler-based approaches such as TVM [11], XLA [28], Ansor [105], TensorIR [22], torch.compile [4], and Hidet [19] focus on automatic fusion via scheduling or tuning. Graph-level optimization frameworks like TASO [35], PET [89], and Automatic Horizontal Fusion [41] perform rule- or cost-based transformations to eliminate redundant computation and improve fusion opportunities. Inference-oriented systems such as DNNFusion [65], ASPEN [72], AStitch [109], TensorRT [70], ONNXRuntime [60], Rammer [54], and Roller [111] use various fusion strategies to optimize runtime performance. Manual approaches like Triton [84] let developers implement custom fused kernels with fine-grained control. Recently, Mirage [95] introduces a multi-level superoptimizer that automatically fuses complex tensor program blocks and shows its benefits for LoRA serving. However, it does not yet address LoRA fine-tuning scenarios with a sufficient number of tokens, dropout, backward computation, or fusion challenges from multi-LoRA kernel execution.

Parallelism and Distributed Training. A lot of work has been done on parallelizing the training of large models, such as Data Parallelism [1, 42], Sharded Data Parallelism [73–75, 104], Tensor Parallelism [81], and Pipeline Parallelism [21, 33, 62]. Hybrid parallelism is usually used to combine multiple parallelism strategies to achieve better performance [7, 81, 106]. To effectively find the optimal parallelization strategies, systems [49, 51, 53, 59, 88, 106, 112] are proposed to automatically find the best combination of parallelism. These automatic planners are orthogonal to our work because any parallelization strategy they produce can directly benefit from our fused kernels.

9 Conclusion

This paper identifies and addresses two critical performance bottlenecks in LLM LoRA fine-tuning: redundant memory access in LoRA modules and missed optimization opportunities for grouping multiple concurrent LoRA jobs. Our solution, LoRAFusion, introduces a novel horizontal fusion technique tailored for LoRA kernels that reduces memory traffic by up to 37% and a complementary job-level scheduling strategy that improves GPU utilization from 65% to 89%. Combined, these optimizations achieve up to 1.96× speedup compared to the state-of-the-art systems across various models and datasets. We hope LoRAFusion will help improve the accessibility and efficiency of LLM LoRA fine-tuning for both researchers and practitioners.

10 Acknowledgement

We sincerely thank our shepherd, Matthias Boehm, and the anonymous reviewers for their valuable feedback. We also appreciate members of the EcoSystem Research Laboratory at the University of Toronto for their discussions and suggestions, with special thanks to Yu Bo Gao and Xiao Zhang for their contributions. The authors with the University of Toronto are supported by Vector Institute Research grants, the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

A Artifact Appendix

A.1 Abstract

We provide the source code of LoRAFusion and scripts to reproduce the major experimental results from the paper. The artifact enables reproduction of key evaluation figures including data distribution analysis (Figure 13), end-to-end performance comparisons (Figure 14), kernel-level performance analysis (Figure 17), layer-wise performance evaluation (Figure 18), and memory traffic analysis (Figure 19). The artifact includes detailed installation procedures and automated evaluation workflows. Full reproduction requires a Linux system with 192 GB RAM, 256 GB disk space, and 4 NVIDIA H100 GPUs with NVLink interconnects. Kernel and layer-level benchmarks can be executed on systems with a single GPU.

A.2 Description & Requirements

A.2.1 How to access. The code is available at: Github https://github.com/CentML/lorafusion and Zenodo https://zenodo.org/records/17051801.

A.2.2 Hardware dependencies. The complete experimental evaluation requires a Linux system equipped with at least 192 GB of system memory, 256 GB of available disk storage, and 4 NVIDIA H100 GPUs interconnected via NVLink.

For partial evaluation, kernel and layer-level benchmarks can be executed on systems with a single GPU. Performance results on alternative hardware configurations may differ from those reported in the paper. Systems with higher compute-to-memory bandwidth ratios typically yield superior performance, while older hardware with lower ratios may exhibit reduced performance gains.

A.2.3 Software dependencies. The artifact requires Conda for environment management. The software stack includes CUDA 12.4, PyTorch v2.6.0, megatron-core v0.11.0, and Triton v3.2.0. All dependencies are automatically installed through the provided setup scripts.

A.2.4 Benchmarks. None

A.3 Set-up

1. Clone the GitHub repository.

```
git clone
  https://github.com/CentML/lorafusion.git
cd lorafusion
git checkout eurosys-ae
```

2. Install the requirements by running this command or following `docs/installation.md`.

```
conda create -y -n lorafusion python=3.12
conda activate lorafusion
cd benchmarks_paper
bash scripts/setup/setup_env.sh
```

3. Download the Hugging Face models and datasets. Make sure you are logged in and have access to them.

```
# huggingface-cli login
python prepare_models.py
python gen_sample_distribution.py
```

4. Verify hardware-specific kernel configurations. The Triton kernels require hardware-specific tuning to optimize tiling strategies. Examine `lorafusion/ops/triton_ops/config.py` to determine if pre-tuned configurations exist for the target hardware. Pre-configured settings are available for:
   - NVIDIA H100 80GB HBM3 (recommended)
   - NVIDIA A100 SXM4 80GB
   - NVIDIA A100 PCIe 80GB
   - NVIDIA GeForce RTX 3090

   For unsupported hardware configurations, execute the kernel tuning process:

```
cd /PATH/TO/lorafusion/
python tools/tune_kernels.py
```

   Then, update `lorafusion/ops/triton_ops/config.py` with the generated optimal configurations.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): LoRAFusion is up to 1.96× faster (average 1.47×) than Megatron-LM, and up to 1.46× faster (average 1.29×) than mLoRA. See Section 6.2 and Figure 14.
- (C2): Our fused kernels are up to 1.39× faster (average 1.27×) and can replace existing LoRA kernels. See Section 6.4 and Figure 17, Figure 18, and Figure 19.

A.4.2 Complete Experimental Evaluation. The full experimental evaluation requires 4 NVIDIA GPUs, each with 80GB memory capacity.

1. Navigate to the `benchmarks_paper` directory.
2. Execute the complete evaluation suite:

```
bash scripts/run_all.sh all
```

   a. The complete evaluation encompasses all primary experiments and kernel performance assessments, requiring approximately 4 hours of computation time.
   b. Detailed command specifications and timing estimates are available in `scripts/run_all.sh`.
   c. Individual experiment subsets can be executed by modifying the script parameters.
3. Evaluation results are generated in the `results` directory, producing figures corresponding to Figure 13, Figure 14, Figure 17, Figure 18, and Figure 19.

A.4.3 Reduced-Scale Evaluation. For systems with limited GPU resources, kernel and layer-level benchmarks can be executed on a single GPU configuration.

- Systems with GPUs containing 80GB or greater memory capacity can execute the comprehensive single-GPU evaluation suite:

```
bash scripts/run_all.sh all_single_gpu
```

- Systems with GPUs containing less than 80GB memory can execute kernel and layer benchmarks independently:

```
bash scripts/run_all.sh layer
bash scripts/run_all.sh kernel
```

Results are generated in the `results` directory. For 80GB+ configurations, the evaluation produces a subset of Figure 14 alongside Figure 17, Figure 18, and potentially Figure 19 (contingent on NCU profiling availability).

A.5 Notes on Reusability

Experimental customization can be achieved by modifying `scripts/run_all.sh` and associated sub-scripts. The artifact provides evaluation scripts and corresponding visualization tools for result generation.

Performance characteristics on alternative GPU architectures may differ from H100-based results. Systems with lower compute-to-memory bandwidth ratios typically exhibit reduced performance gains. Power consumption constraints during kernel configuration tuning may affect optimal parameter selection and subsequent benchmark accuracy. For consistent performance evaluation across different hardware, manual GPU frequency configuration is recommended:

```
# Disable automatic frequency scaling
sudo nvidia-smi -pm 1
sudo nvidia-smi --auto-boost-default=0

# Query supported frequency configurations
nvidia-smi -q -d SUPPORTED_CLOCKS

# Configure specific memory and graphics clock frequencies
# sudo nvidia-smi -ac <memory_clock,graphics_clock>
# e.g.,
# sudo nvidia-smi -ac 6251,1050
```

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, David Berard, Geeta Chauhan, Anjali Chourdia, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. (2024).

[5] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices* 50, 8 (2015), 173–182.

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[7] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid parallelization strategies for large-scale machine learning in systemml. *Proceedings of the VLDB Endowment* 7, 7 (2014), 553–564.

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

[9] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems* 6 (2024), 1–13.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*.

[12] Nachshon Cohen, Oren Kalinsky, Yftah Ziser, and Alessandro Moschitti. 2021. Wikisum: Coherent summarization dataset for efficient human-evaluation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 212–219.

[13] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*.

[14] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems* 36 (2023), 10088–10115.

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

[16] Ning Ding, Xingtai Lv, Qiaosen Wang, Yulin Chen, Bowen Zhou, Zhiyuan Liu, and Maosong Sun. 2023. Sparse low-rank adaptation of pre-trained language models. *arXiv preprint arXiv:2311.11696* (2023).

[17] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Yu Hao, Yida Wang, and Gennady Pekhimenko. 2025. Tilus: A Virtual Machine for Arbitrary Low-Precision GPGPU Computation

in LLM Serving. *arXiv preprint arXiv:2504.12984* (2025).

[18] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[19] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 370–384.

[20] facebookresearch/llama. 2024. llama3. https://github.com/meta-llama/llama3.

[21] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[22] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.

[23] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning. (2020).

[24] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28 (2015)*. 2962–2970.

[25] Fireworks. 2024. Fireworks/fine-tuning. https://docs.fireworks.ai/fine-tuning/fine-tuning-models.

[26] Chao Gao and Sai Qian Zhang. 2024. Dlora: Distributed parameter-efficient fine-tuning solution for large language model. *arXiv preprint arXiv:2404.05182* (2024).

[27] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. Ai and memory wall. *IEEE Micro* 44, 3 (2024), 33–39.

[28] Google. 2022. XLA. https://www.tensorflow.org/xla.

[29] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*. PMLR, 3929–3938.

[30] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International conference on machine learning*. PMLR, 2790–2799.

[31] Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. 2024. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989* (2024).

[32] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[33] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[34] Adam Ibrahim, Benjamin Thérien, Kshitij Gupta, Mats L Richter, Quentin Anthony, Timothée Lesort, Eugene Belilovsky, and Irina Rish. 2024. Simple and scalable strategies to continually pre-train large language models. *arXiv preprint arXiv:2403.08763* (2024).

[35] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[36] Ishan Jindal, Chandana Badrinath, Pranjal Bharti, Lakkidi Vinay, and Sachin Dev Sharma. 2024. Balancing Continuous Pre-Training and Instruction Fine-Tuning: Optimizing Instruction-Following in LLMs. *arXiv preprint arXiv:2410.10739* (2024).

[37] Dawid J Kopiczko, Tijmen Blankevoort, and Yuki M Asano. 2023. Vera: Vector-based random matrix adaptation. *arXiv preprint arXiv:2310.11454* (2023).

[38] Achintya Kundu, Rhui Dih Lee, Laura Wynter, Raghu Kiran Ganti, and Mayank Mishra. 2024. Enhancing training efficiency using packing with flash attention. *arXiv preprint arXiv:2407.09105* (2024).

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

[40] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[41] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic horizontal fusion for GPU kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 14–27.

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

[43] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease. ml: Towards multi-tenant resource sharing for machine learning workloads. *Proceedings of the VLDB Endowment* 11, 5 (2018), 607–620.

[44] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).

[45] Yixiao Li, Yifan Yu, Chen Liang, Pengcheng He, Nikos Karampatziakis, Weizhu Chen, and Tuo Zhao. 2023. Loftq: Lora-fine-tuning-aware quantization for large language models. *arXiv preprint arXiv:2310.08659* (2023).

[46] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*. PMLR, 6543–6552.

[47] Sheng Lin, Fangcheng Fu, Haoyang Li, Hao Ge, Xuanyu Wang, Jiawen Niu, Yaofeng Tu, and Bin Cui. 2025. LobRA: Multi-tenant Fine-tuning over Heterogeneous Data. *arXiv preprint arXiv:2509.01193* (2025).

[48] Zhaojiang Lin, Andrea Madotto, and Pascale Fung. 2020. Exploring versatile generative language model via parameter-efficient transfer learning. *arXiv preprint arXiv:2004.03829* (2020).

[49] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. 2024. {nnScaler}:{Constraint-Guided} Parallelization Plan Generation for Deep Learning Training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 347–363.

[50] Dongqi Liu and Vera Demberg. 2024. Rst-lora: A discourse-aware low-rank adaptation for long document abstractive summarization. *arXiv preprint arXiv:2405.00657* (2024).

[51] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. 2024. Aceso: Efficient Parallel DNN Training through Iterative Bottleneck Alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 163–181.

[52] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. 2024. Dora: Weight-decomposed low-rank adaptation. In *Forty-first International Conference on Machine Learning*.

[53] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 553–564.

[54] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

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

[56] meta llama. 2024. llama3.1. https://ai.meta.com/blog/meta-llama-3-1/.

[57] meta llama. 2025. meta-llama/llama-cookbook. https://github.com/meta-llama/llama-cookbook.

[58] meta llama/llama3. 2024. llama/MODEL_CARD.md. https://github.com/meta-llama/models/llama3_1/blob/main/MODEL_CARD.md.

[59] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2023. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism. *Proc. VLDB Endow.* 16, 3 (2023), 470–479. doi:10.14778/3570690.3570697

[60] Microsoft. 2025. microsoft/onnxruntime. https://github.com/microsoft/onnxruntime.

[61] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium.

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

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

[64] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[65] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[66] NVIDIA. 2008. InfiniBand. https://network.nvidia.com/pdf/whitepapers/WP_InfiniBand_Technology_Overview.pdf.

[67] NVIDIA. 2014. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.

[68] NVIDIA. 2025. NVIDIA DGX B200. https://resources.nvidia.com/en-us-dgx-systems/dgx-b200-datasheet.

[69] NVIDIA. 2025. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.

[70] NVIDIA. 2025. NVIDIA/TensorRT. https://github.com/NVIDIA/TensorRT.

[71] OpenAI. 2024. OpenAI/openai-fine-tuning. https://platform.openai.com/docs/guides/fine-tuning.

[72] Jongseok Park, Kyungmin Bin, Gibum Park, Sangtae Ha, and Kyunghan Lee. 2023. Aspen: Breaking operator barriers for efficient parallelization of deep neural networks. *Advances in Neural Information Processing Systems* 36 (2023), 68625–68638.

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

[74] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

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

[76] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. 2020. Adapterdrop: On the efficiency of adapters in transformers. *arXiv preprint arXiv:2010.11918* (2020).

[77] RunPod. 2025. RunPod/fine-tuning. https://docs.runpod.io/fine-tune/.

[78] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).

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

[80] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 701–718.

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

[82] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. 2015. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 368–380.

[83] Shubo Tian, Qiao Jin, Lana Yeganova, Po-Ting Lai, Qingqing Zhu, Xiuying Chen, Yifan Yang, Qingyu Chen, Won Kim, Donald C Comeau, et al. 2024. Opportunities and challenges for ChatGPT and large language models in biomedicine and health. *Briefings in Bioinformatics* 25, 1 (2024), bbad493.

[84] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[85] Together.AI. 2024. Together.AI/fine-tuning. https://docs.together.ai/reference/finetune.

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

[87] Christophe Tribes, Sacha Benarroch-Lelong, Peng Lu, and Ivan Kobyzev. 2023. Hyperparameter optimization for large language model instruction-tuning. *arXiv preprint arXiv:2312.00949* (2023).

[88] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.

[89] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 37–54.

[90] Shuhe Wang, Guoyin Wang, Yizhong Wang, Jiwei Li, Eduard Hovy, and Chen Guo. 2024. Packing analysis: Packing is more appropriate for large models or datasets in supervised fine-tuning. *arXiv preprint arXiv:2410.08081* (2024).

[91] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. *Proceedings of Machine Learning and Systems* 3 (2021), 599–623.

[92] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560* (2022).

[93] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. Association for Computational Linguistics, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[94] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. {dLoRA}: Dynamically orchestrating requests and adapters for {LoRA}{LLM} serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 911–927.

[95] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2024. Mirage: A Multi-Level Superoptimizer for Tensor Programs. *arXiv preprint arXiv:2405.05751* (2024).

[96] Guangxuan Xiao, Ji Lin, and Song Han. 2023. Offsite-tuning: Transfer learning without full model. *arXiv preprint arXiv:2302.04870* (2023).

[97] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115* (2024).

[98] Zhengmao Ye, Dengchun Li, Zetao Hu, Tingfeng Lan, Jian Sha, Sicong Zhang, Lei Duan, Jie Zuo, Hui Lu, Yuanchun Zhou, et al. 2023. mLoRA: Fine-Tuning LoRA Adapters via Highly-Efficient Pipeline Parallelism in Multiple GPUs. *arXiv preprint arXiv:2312.02515* (2023).

[99] Jiseon Yun, Jae Eui Sohn, and Sunghyon Kyeong. 2023. Fine-tuning pretrained language models to enhance dialogue summarization in customer service centers. In *Proceedings of the Fourth ACM International Conference on AI in Finance*. 365–373.

[100] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Yang You, Guiming Xie, Xuejian Gong, and Kunlong Zhou. 2025. Train Small, Infer Large: Memory-Efficient LoRA Training for Large Language Models. *arXiv preprint arXiv:2502.13533* (2025).

[101] Kai Zhang, Rong Zhou, Eashan Adhikarla, Zhiling Yan, Yixin Liu, Jun Yu, Zhengliang Liu, Xun Chen, Brian D Davison, Hui Ren, et al. 2024. A generalist vision–language foundation model for diverse biomedical tasks. *Nature Medicine* (2024), 1–13.

[102] Bowen Zhao, Hannaneh Hajishirzi, and Qingqing Cao. 2024. Apt: Adaptive pruning and tuning pretrained language models for efficient training and inference. *arXiv preprint arXiv:2401.12200* (2024).

[103] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* 1, 2 (2023).

[104] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch FSDP: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[105] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[106] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[107] Ying Zheng, Lei Jiao, Han Yang, Lulu Chen, Ying Liu, Yuxiao Wang, Yuedong Xu, Xin Wang, and Zongpeng Li. 2024. Online Scheduling and Pricing for Multi-LoRA Fine-Tuning Tasks. In *Proceedings of the 53rd International Conference on Parallel Processing*. 357–366.

[108] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. LLaMaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Association for Computational Linguistics, Bangkok, Thailand. http://arxiv.org/abs/2403.13372

[109] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.

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

[111] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.

[112] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorveedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. 2025. Mist: Efficient Distributed Training of Large Language Models via Memory-Parallelism Co-Optimization. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1298–1316.

[113] Weiming Zhuang, Yonggang Wen, and Shuai Zhang. 2022. Smart multi-tenant federated learning. *arXiv preprint arXiv:2207.04202* (2022).