ML-TOMB: Multi-LoRA Training with One-Model Base

Renjie Zhao
University of Washington
Seattle, Washington, USA
renjiz2@cs.washington.edu

Alek Curless
University of Washington
Seattle, Washington, USA
alekcurl@cs.washington.edu

Abstract

As foundation models built on transformer architectures [10] continue to advance across domains such as text generation [10], computer vision [9], image generation [2, 4], and robotic control [5], downstream fine-tuning increasingly relies on parameter-efficient methods like LoRA [3]. Many personalization or small-data tasks [6, 7] use only a few training samples, forcing extremely small batch sizes that fails to utilize all GPU memories. In these settings, most of the GPU memory is consumed by the base model itself, while the incremental LoRA parameters occupy only a tiny fraction, making single-adapter training highly memory-inefficient.

To better leverage available compute, we investigate training multiple LoRA adapters simultaneously on a shared base model. However, naïvely doing this in PyTorch requires running separate linear projections for each adapter, producing many small GEMM operations during both forward and backward passes, which leads to substantial kernel-launch overhead and poor scalability. Prior work [1] addresses this issue only for *inference*, introducing an SGMV operator that merges multiple LoRA forward paths into a single batched kernel. Yet it does not support training and assumes all adapters modify the same set of linear layers—an assumption incompatible with real-world fine-tuning pipelines where different adapters often target different subsets of the model.

To address these issues, we introduce ML-TOMB, which extends the current methods in two key ways: (1) each adapter may target a different subset of linear layers, and inputs may optionally bypass LoRA; and (2) we add an SGMV-based backward operator that enables efficient multi-LoRA *training*, not just inference. Our method preserves numerical equivalence to standard per-adapter training while substantially improving throughput and resource utilization.

Our implementation is visiable to the course staffs at https://gitlab.cs.washington.edu/renjiz2/599o-project.

1 Introduction

Large transformer-based foundation models have become the default backbone for a wide range of downstream tasks. In practice, parameter-efficient fine-tuning methods such as LoRA [3] are widely used to adapt these models, especially in personalization, small-data, or domain-specific settings. Such tasks often contain only a few training samples, forcing extremely small batch sizes [6, 7]. In these scenarios, the base model dominates GPU memory consumption while the LoRA parameters themselves are small, making single-adapter training inefficient and motivating the need to train multiple LoRA adapters on a shared base model.

In this work, we focus on the problem of *multi-LoRA training*: executing the forward and backward passes of several LoRA adapters within a single model invocation. Formally, given a frozen base model and a collection of LoRA modules that may attach to different linear layers, our goal is to train all adapters jointly while preserving the numerical equivalence of training each adapter independently. A key requirement in realistic fine-tuning pipelines is that different batch elements may activate different sets of LoRA adapters; the system must therefore route each sample through its designated adapters and ensure that their parameter updates remain completely isolated. An effective solution should minimize redundant computation and support flexible adapter configurations where different tasks modify different subsets of the model.

However, naïvely implementing multi-LoRA training in PyTorch is inefficient. Each adapter introduces an additional low-rank update to the affected linear layers, which results in multiple small GEMM operations during both the forward and backward passes. These operations incur significant kernel-launch overhead and scale poorly as the number of adapters increases. Prior work such as Punica [1] addresses this inefficiency only for *inference*, using a Segmented Gather Matrix-Vector Multiplication (SGMV) operator to merge multiple LoRA forward paths into a single batched kernel. Yet Punica does not support training, provides no mechanism for batching backward passes, and further assumes that all LoRA adapters modify the same set of linear layers—an assumption that does not hold in real-world fine-tuning pipelines, where adapters often attach to different layers or may be selectively enabled per input.

To address these limitations, we propose **ML-TOMB**, a proof-of-concept framework that extends Punica's SGMV design to support multi-LoRA training. Rather than aiming to be a drop-in replacement or production-ready system, our goal is to demonstrate that multi-LoRA training can be made both correct and efficient when the base model forward and backward passes are shared. ML-TOMB allows each LoRA module to target an arbitrary subset of linear layers, and also supports inputs that bypass LoRA entirely during inference, enabling more flexible adapter configurations.

On the systems side, we incorporate SGMV into the backward pass by accelerating the computation of grad_v =

grad_output · B using a single batched kernel. The remaining gradients are still computed using per-adapter matrix multiplications. Despite this limitation, our implementation maintains numerical equivalence to independently training each adapter and reduces a significant portion of redundant GEMM operations. The resulting system provides initial evidence that multi-LoRA training is feasible and can substantially reduce resource overhead in small-data regimes.

2 Background

2.1 LoRA

Low-Rank Adaptation (LoRA) [3] is an efficient parameter-update technique that augments selected linear layers with a low-rank residual. For a pretrained weight matrix $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$, LoRA introduces two trainable matrices $A \in \mathbb{R}^{r \times d_{out}}$ and $B \in \mathbb{R}^{d_{\text{in}} \times r}$, whose product BA forms a rank-r update while W is kept frozen:

$$W_{\text{LoRA}} = BA \tag{1}$$

$$y = xW_0 + xW_{\text{LoRA}} \tag{2}$$

Due to the low rank of the trainable matrices, the number of parameters that need to be trained is significantly lower than the parameters of the pretrained parameters (by 2-3 orders of magnitude depending on rank). This makes LoRA ideal for efficient downstream fine-tuning, with end results comparable to full model fine-tuning. At inference time, the adapter weights can be added either directly to the base model weights with no performance cost, or kept separate for fast switching to just base model inference or to swap with other adapters at a small increase to latency.

2.2 Related Work

2.2.1 Punica. The system proposed by *Punica: Multi-Tenant LoRA Serving* [1] first exploits the idea of serving many LoRA-fine-tuned models share the same pretrained backbone to enable efficient multi-tenant deployment. It introduces a custom SGMV kernel that batches the adapter computations of multiple, different LoRA models, so a single GPU can serve many LoRA variants without replicating the base model. By consolidating diverse LoRA requests into large batched GPU workloads and using on-demand adapter loading, Punica achieves much higher throughput compared to naive multi-model serving. However, as shown in Fig. 1, their implementation requires weights of all blocks in a LoRA adapter to be allocated in a contiguous block, and all LoRA adapters are targeting to the same set of layers because it accepts a single `layer_idx` for all LoRA adapters. These limitations make it incompatible with real-life needs: different fine-tuning tasks often target different subsets of layers, and LoRA adapters are rarely aligned in structure.

2.2.2 S-LoRA.. The system from *S-LoRA: Serving Thousands of Concurrent LoRA Adapters* [8] targets scalable serving of very large numbers of LoRA adapters. Rather than merging adapters into base weights, S-LoRA splits base-model computation (batched GEMM) from LoRA adapter computations, executing the latter via specialized custom CUDA kernels that handle heterogeneous adapter ranks and sequence lengths. It also uses a "Unified Paging" memory pool that dynamically loads only the adapters needed by current queries into GPU memory — enabling thousands of adapters to coexist in main memory while only a small subset occupies GPU memory at runtime.

2.3 Motivation

Parameter-efficient fine-tuning methods such as LoRA have become the default strategy for adapting large transformer models. New personalization and per-instance fine tuning approaches like DreamBooth [7] and UltraZoom [6] emerge recently. These approaches trains on extremely few examples, often times less than 5, and therefore uses extremely small batch sizes, resulting the training process oftentimes cannot utilize all available GPU memory.

A natural solution is to train multiple LoRA adapters simultaneously on a shared backbone, thereby amortizing the cost of the base model. However, naïve implementations in PyTorch invoke separate low-rank projections for each adapter and each modified layer, producing many small GEMM operations that suffer from kernel-launch overhead and scale poorly with the number of adapters. Prior work has addressed this inefficiency only in the context of inference and further assumes that all adapters modify an identical set of layers. These constraints do not reflect real fine-tuning pipelines, where adapters frequently target heterogeneous subsets of layers, motivating the need for a flexible and efficient multi-LoRA training mechanism.

3 Design

In this section, we first discuss our modification of the SGMV kernel and then our method of using it during training.

3.1 SGMV Kernel

Formally, given a flattened 2d tensors x and y, we define the calculation of the SGMV kernel to be:

$$SGMV(y|x, s, e, W)[s_i : e_i] = y[s_i : e_i] + x[s_i : e_i]W_i \tag{3}$$

Where i is the index of the LoRA block, s and e are the start and end of the batch elements in the sequence dimension that goes to ith LoRA block.

As proposed by Punica [1], we can use two copies of the SGMV kernel to calculate the forward pass for each layer:

$$v = \mathbf{0} \tag{4}$$

$$v \leftarrow SGMV(v|x, s, e, A) \tag{5}$$

$$y \leftarrow SGMV(y|v, s, e, B) \tag{6}$$

The SGMV kernel accelerates multi-LoRA computation by merging many small matrix multiplications into a single

Original Punica SGMV Signature

```
bool sgmv(
  DType *y,          // the output tensor
  DType *x,          // the input tensor
  DType **w,         // pointer to all weight
      tensor of each LoRA adapter
  int32_t *s,        // start of each group,
      len(s) = num_problems + 1
  void *tmp_d,       // temporary device
      buffer
  int num_problems,  // number of lora
  int d_in,          // input dimension
  int d_out,         // output dimension
  int layer_idx,     // layer index to
      identify
  cudaStream_t stream
);
```

Our Modified SGMV Signature

```
bool sgmv(,
  DType *y,          // the output tensor
  DType *x,          // the input tensor
  DType **w,         // pointer to weight
      tensor of current layer
  int32_t *s,        // start of each group,
      len(s) = num_problems
  int32_t *e,        // end of each group
  void *tmp_d,       // temporary device
      buffer
  int num_problems,  // number of lora
      adapters used in this layer
  int d_in,          // input dimension
  int d_out,         // output dimension
  cudaStream_t stream
);
```

Figure 1. Comparison between original Punica SGMV signature and our modified version.

grouped GEMM operation. In a naïve implementation, each adapter requires computing a tiny GEMM $x[s_i : e_i]W_i$, and launching these kernels individually incurs high overhead and poor GPU utilization. SGMV instead uses CUTLASS's `GemmGrouped` to execute all GEMMs in one batched kernel. This allows the hardware to load-balance across various matrix sizes, amortizes launch overhead, and keeps the GPU fully occupied. In our work, we retain Punica's grouped-GEMM design and only modify the preprocessing logic to support flexible routing and training scenarios.

As discussed in Sec. 2.2.1, the original SGMV is imcompatable with our goal of flexibility. To solve this problem, we changed the function signature in the following ways:

- **Per-adapter weight pointers instead of a global tensor.** In Punica, `ptr_w[i]` is computed as an offset into a large global tensor using `layer_idx`: `w[i] + layer_idx * d_in * d_out`. In our version, `w` directly point to the per-layer weight pointer for each adapter. This simplifies integration with our PyTorch modules and removes the need for a global weight layout.
- **Explicit support for "skipping" tokens via `e`.** Punica assumes that every row in `x` participates in some LoRA update and encodes group boundaries using a single array `s` with `m = s[i+1] - s[i]`, where `m` is the number of rows that participate in adapter i's multiplication. This contradict with our goal of flexibility. If a LoRA adapter does not target a layer, ML-TOMB should automatically skips that adapter for all batch elements that selected it, without corrupting routing or applying incorrect updates. To support our flexible design, we introduce a separate end-index array `e` and compute `m = e[i] - s[i]`, which allows some rows of `x` to bypass LoRA of this layer.

The detailed comparison between the original Punica SGMV signature and our modified version is shown in Fig. 1.

3.2 Training with SGMV

Backpropagation through a multi-LoRA layer requires computing gradients for v, A_i, B_i, and x, where $v = SGMV(v|x, s, e, A)$ and $y = SGMV(y|v, s, e, B)$. These yield $g^v = g^y B_i$, $\nabla B_i = (g^y[s_i : e_i])^\top v[s_i : e_i]$, $g^x = g^v A_i$, and $\nabla A_i = (g^v[s_i : e_i])^\top x[s_i : e_i]$.

Using SGMV for g^v. We realize the term $g^v[s_i : e_i] = g^y[s_i : e_i]B_i$ has the same row-partitioned structure as the forward LoRA projection. Each token slice routes to exactly one adapter and is multiplied by its corresponding B_i. Therefore, all g^v computations can be fused into a single grouped GEMM using the same (s, e) segmentation as the forward pass.

Why $\nabla B, g^x$, and ∇A cannot use SGMV. However, $\nabla B_i = (g^y[s_i : e_i])^\top v[s_i : e_i]$ and $\nabla A_i = (g^v[s_i : e_i])^\top x[s_i : e_i]$ are both reductions over tokens that produce stand-alone parameter matrices, which cannot be written into token-aligned row ranges as required by SGMV. Similarly, $g^x[s_i : e_i] = g^v[s_i : e_i]A_i$ requires accumulation into a shared tensor, conflicting with SGMV's overwrite-only, row-partitioned execution model. Therefore, we fall back to computing these terms using inexpensive per-adapter GEMMs.

4 Implementation

All following implementation is built on top of our implementation of transformer models in HW1.

4.1 LoRA Structure in PyTorch

We represent each per-layer LoRA adapter with a `LoraBlock` class, which simply stores the two low-rank matrices `lora_A` and `lora_B`.

We also implement a `MultiLoraLinear` class as a replacement of the `Linear` class that allows multiple `LoraBlocks` be attached. Each `MultiLoraLinear` layer maintains a dictionary mapping a globally unique LoRA index to its local `LoraBlock` instance. When `add_lora` is called, the layer allocates a new `LoraBlock` with the appropriate input/output dimensions and rank, and registers it under the given index.

When `add_lora` is invoked on the transformer, the user provides a list of strings specifying which linear layers should receive LoRA adapters. The transformer parses this list, determines the set of target layers, assigns a globally unique `lora_id` to this adapter, and then calls `add_lora` on each selected layer with that global index. Each layer creates and registers its own `LoraBlock` corresponding to the same `lora_id`.

4.2 SGMVFunction

We implement `SGMVFunction`, a `torch.autograd.Function` that invokes the SGMV kernel to support both inference and training. The function exposes a simple interface to PyTorch while allowing the kernel to operate over an arbitrary number of LoRA adapters.

In the **forward pass**, the SGMV kernel replaces multiple independent projections through the LoRA A and B matrices like discussed in Sec. 3.1.

In the **backward pass**, following Sec. 3.2, we use the SGMV kernel to compute $g^v = g^y B$ in a single grouped GEMM. The remaining gradients ∇B, g^x, and ∇A are computed using per-adapter GEMMs.

5 Evaluation

In this section, we first demonstrate the correctness of our modified SGMV kernel comparing to a naïve PyTorch implementation, and then evaluate its performance during both the forward and backward pass of training multiple LoRAs on a single base model. All test cases can be found in https://gitlab.cs.washington.edu/renjiz2/599o-project/-/tree/master/src

5.1 Test Setup

For all conducted tests, we use the Transformer-Large configuration from HW1 for our base model. Additionally, we use a batch size of 16 and a sequence length of 1000. For LoRA adapters, we use a LoRA rank of 8. Unless otherwise specified, apply LoRA to all linear layers in the transformer model, and the load is evenly distributed among all LoRAs. Since the main goal is to show correctness and performance, all the inputs to the model are randomly generated tensors. The test configuration is controlled by the config file

```
(599o-project) [renjiz2@klone-login01 599o-project]$ a40
  ✓ Outputs from SGMV and vanilla match closely!

Pass 26:
  ✓ Outputs from SGMV and vanilla match closely!

Pass 27:
  ✓ Outputs from SGMV and vanilla match closely!

Pass 28:
  ✓ Outputs from SGMV and vanilla match closely!

Pass 29:
  ✓ Outputs from SGMV and vanilla match closely!

Pass 30:
  ✓ Outputs from SGMV and vanilla match closely!

✓ All tests passed!
✓ Model successfully grouped samples by LoRA index and processed them!
```

Figure 2. Correctness of Forward Pass

```
(599o-project) [renjiz2@klone-login01 599o-project]$ a40
  Loss (SGMV):   7.277344
  Loss (Vanilla): 7.277344
  Loss difference: 0.000000e+00
  ✓ All gradients match!

Pass 29:
  Loss (SGMV):   7.250000
  Loss (Vanilla): 7.250000
  Loss difference: 0.000000e+00
  ✓ All gradients match!

Pass 30:
  Loss (SGMV):   7.429688
  Loss (Vanilla): 7.429688
  Loss difference: 0.000000e+00
  ✓ All gradients match!

✓ All tests passed!
✓ Backward pass correctness verified!
(599o-project) [renjiz2@g3050 599o-project]$
```

Figure 3. Correctness of Backward Pass

here: https://gitlab.cs.washington.edu/renjiz2/599o-project/-/blob/master/src/test_configs/test_config_large.json.

For comparison with PyTorch only (vanilla) baseline, we added an argument of `use_sgmv` to our `MultiLoraLinear` module to switch between the two implementations. When this flag is off, the forward and backward pass will be computed using standard PyTorch operations that process each LoRA adapter separately with only PyTorch built-in functions. When the flag is on, our modified SGMV kernel will be used instead. All test are conducted on an NVIDIA A40 GPU with 48GB memory.

5.2 Correctness

To verify correctness, we compare the outputs and gradients produced by our SGMV-based implementation against the vanilla implementation. We construct randomly initialized mixed-LoRA batches where different samples are routed to different adapters, run both forward passes with `use_sgmv` on and off, and assert that the logits match each other. For training, we perform a stricter backward-to-backward comparison. After running `loss.backward()`, we save all LoRA gradients, then recompute the loss using the vanilla implementation and run a second backward pass. As shown in

```
(599o-project) [renjiz2@klone-login01 599o-project]$ a40
', 'layers.30.attention.to_out', 'layers.30.ffn.w1', 'layers.31.attention.to_q',
, 'layers.32.attention.to_k', 'layers.32.attention.to_v', 'layers.32.attention.t
attention.to_k', 'layers.33.attention.to_v', 'layers.33.attention.to_out', 'laye
ers.34.attention.to_k', 'layers.34.attention.to_v', 'layers.34.attention.to_out'
rs.35.attention.to_k', 'layers.35.attention.to_v', 'layers.35.attention.to_out',

Model parameters after LoRA: 971,981,480
Number of LoRA modules: 4

Freezing base model and enabling LoRA gradients...
Trainable parameters: 0

Creating fake input with mixed LoRA indices...
LoRA indices: [0, 0, 1, 1, 2, 2, 3, 3, 4, 4]

Running forward passes with automatic LoRA grouping...

✓ All tests passed!
✓ Model successfully grouped samples by LoRA index and processed them!
(599o-project) [renjiz2@g3050 599o-project]$
```

Figure 4. Correctness of Our Flexible LoRA Design using SGMV Kernel

our test harness, we evaluate per-parameter differences and require all LoRA gradients to match.

As shown in Fig. 2 and Fig. 3. Across all evaluation passes, the SGMV and vanilla implementations produce identical logits, losses, and parameter gradients, with maximal deviations well within the expected fp16 numerical range. Therefore, our SGMV kernel and PyTorch integration faithfully reproduce the behavior of the standard multi-LoRA computation while providing substantially improved performance.

5.3 Flexibility

To evaluate the flexibility of our design, we test whether SGMV remains correct when each LoRA adapter can attach to a different set of linear layers. In our test harness, every LoRA adapter independently selects roughly 75% of all linear layers at random.

Similar to Sec. 5.2, for each configuration, we construct batches with mixed LoRA indices and run forward passes using both the SGMV-enabled and vanilla implementations. We assert that the logits produced by the two implementations match each other. This verifies that SGMV correctly routes tokens to their assigned LoRA adapters even when different adapters activate different subsets of layers,and when the set of active LoRA modules changes from layer to layer. As shown in Fig. 4, across all iterations, our tests report no mismatches, demonstrating that the grouped execution remains numerically equivalent regardless of adapter placement.

5.4 Forward Performance

Now we evaluate the full-scale model during the forward pass of the SGMV, performing 10 warm-up steps and then averaging the total step time across the next 100 inference steps. The results are shown in Fig. 5 For just one LoRA, the SGMV starts with 117.4ms per step, only marginally worse than the baseline's 111.2ms. The step per pass barely creeps up for SGMV with shallow linear scale, at no more than about a couple milliseconds per LoRA. The baseline also scales linearly but much more steeply, at about 37ms

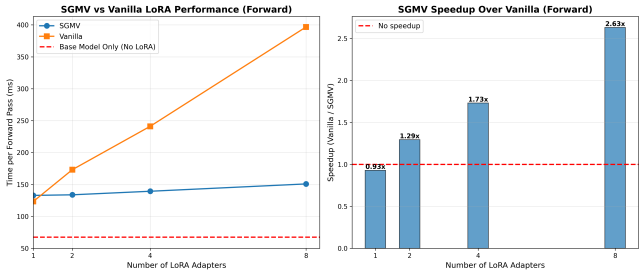

Figure 5. Evaluation of the forward pass performance of the transformer with different number of LoRA adapters using both the baseline and SGMV implementations.

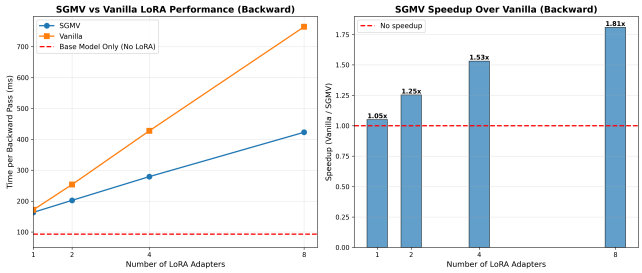

Figure 6. Evaluation of the backward pass performance of the transformer with different number of LoRA adapters using both the baseline and SGMV implementations.

per step per LoRA. So, from just 2 LoRAs, there is already a significant improvement.

The exact speed up ratio is shown for further comparison, but clearly most striking is the forward SGMV's ability to handle 8 LoRAs at 2.89x the speed of the baseline, at 131ms per step rather than 369ms.

5.5 Backward Performance

Next, we examine the backward SGMV, again performing 10 warm-up steps and averaging the next 100 backward passes (Fig. 6). As expected for the increased memory I/O pressure, operations, and kernel dispatches of gradient calculations, the backward SGMV takes 144.5ms per step, which is marginally better than the baseline's 148.8ms. The backward SGMV incurs an increased penalty compared to the forwards pass at about 34ms per additional LoRA, but is still much improved over the baseline's 73ms per additional LoRA.

Despite the only being able to use the kernel once in the backward pass, we still see a 1.76x speedup over the baseline when juggling 8 LoRAs, or 388ms per SGMV step compared to 665ms.

5.6 Microbenchmark

To give context for the performance measured in later benchmarks, we start with a microbenchmark comparing just the kernel's performance to the same compute done with

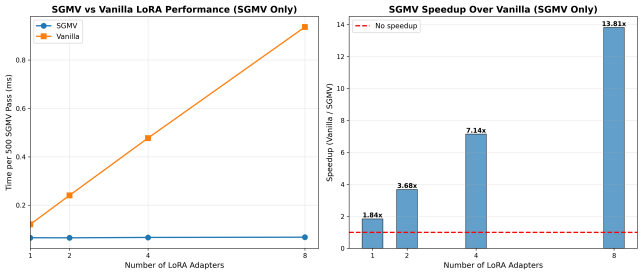

Figure 7. Evaluation of the SGMV kernel and same computation using PyTorch only.

PyTorch-only implementation. We run 500 passes of each, measuring the section which performs computation of Eq. 5 and Eq. 6, then average the samples. Then we increase the number of LoRAs and decrease the per LoRA batch size and repeat. The results are demonstrated in Fig. 7.

As demonstrated in Punica, the latency of the kernel itself stays essentially constant as we trade off LoRA quantity and per LoRA batch size (approximately 64µs). Since the baseline has to fire off separate CUDA streams proportional to the number of LoRAs, each consisting of many queued kernels, launch overhead and large tensors stall the effective parallelism. The result is terrible, near linear scaling of throughput for the baseline, giving the kernel a 14.15x speedup.

6 Discussion

6.1 Limitation

While our approach demonstrates substantial improvements for multi-LoRA inference and partial acceleration during training, it also introduces several inherent limitations.

First, the current system invokes the SGMV kernel only once in each backward pass, which restricts the achievable speedup during training. Although the kernel effectively fuses the dominant gradient projection through the `grad_v` matrices, other gradient components continue to rely on unfused per-adapter operations. This asymmetry limits the end-to-end acceleration of the training pipeline. A promising avenue for future work is the development of complementary SGMV-style kernels for the remaining gradient paths, enabling a fully fused backward pass that mirrors the efficiency of the forward computation.

Second, our implementation requires that all LoRA adapters attached to the same layer share a uniform rank. This constraint arises because SGMV constructs a single grouped GEMM with fixed rank, and variable adapter ranks would lead to irregular segment boundaries and poorly balanced computation. While in principle one could extend the kernel to accept per-adapter rank metadata and segmented `x` and `y` tensors, such flexibility would complicate the batching logic and likely diminish the performance benefit due to reduced GEMM regularity and increased kernel-launch overhead. Investigating kernel designs that support variable ranks while preserving high arithmetic intensity remains an open challenge.

Overall, removing these constraints—full backward fusion and variable-rank support—could be important directions for broadening the practicality and generality of SGMV-based multi-LoRA training.

6.2 Future Work

Several extensions can further improve the flexibility and efficiency of our framework. First, supporting variable LoRA ranks within the same layer would make the system more broadly applicable. A practical design is to introduce a fallback mechanism: when all adapters share the same rank, the current high-performance SGMV kernel is used; otherwise, the system switches to a more general variable-rank path. This preserves optimal speed in the common case while enabling broader configurability.

Second, developing dedicated backward-acceleration kernels—mirroring the grouped computation used in the forward pass—would enable full end-to-end SGMV speedup during training. Such kernels could fuse additional gradient paths beyond the current single matrix projection, further reducing per-adapter overhead and narrowing the gap between training and inference performance.

7 Conclusion

We presented ML-TOMB, a proof-of-concept system that enables efficient multi-LoRA training by extending the SGMV operator to support both flexible adapter placement and a batched backward pass. By modifying the SGMV interface and integrating it into PyTorch, our system preserves numerical equivalence to standard per-adapter training while eliminating redundant small GEMM operations. Evaluation results show that ML-TOMB maintains correctness across variable LoRA configurations and provides substantial speedups in both forward and backward computation. Together, these findings demonstrate that multi-LoRA training is not only feasible but also highly efficient when the base model's computation is shared, opening the door for more scalable and resource-friendly fine-tuning workflows.

References

[1] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2023. Punica: Multi-Tenant LoRA Serving. arXiv:2310.18547 [cs.DC] https://arxiv.org/abs/2310.18547

[2] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, Kyle Lacey, Alex Goodwin, Yannik Marek, and Robin Rombach. 2024. Scaling Rectified Flow Transformers for High-Resolution Image Synthesis. arXiv:2403.03206 [cs.CV] https://arxiv.org/abs/2403.03206

[3] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL]

https://arxiv.org/abs/2106.09685

[4] Black Forest Labs, Stephen Batifol, Andreas Blattmann, Frederic Boesel, Saksham Consul, Cyril Diagne, Tim Dockhorn, Jack English, Zion English, Patrick Esser, Sumith Kulal, Kyle Lacey, Yam Levi, Cheng Li, Dominik Lorenz, Jonas Müller, Dustin Podell, Robin Rombach, Harry Saini, Axel Sauer, and Luke Smith. 2025. FLUX.1 Kontext: Flow Matching for In-Context Image Generation and Editing in Latent Space. arXiv:2506.15742 [cs.GR] https://arxiv.org/abs/2506.15742

[5] Jason Lee, Jiafei Duan, Haoquan Fang, Yuquan Deng, Shuo Liu, Boyang Li, Bohan Fang, Jieyu Zhang, Yi Ru Wang, Sangho Lee, Winson Han, Wilbert Pumacay, Angelica Wu, Rose Hendrix, Karen Farley, Eli VanderBilt, Ali Farhadi, Dieter Fox, and Ranjay Krishna. 2025. MolmoAct: Action Reasoning Models that can Reason in Space. arXiv:2508.07917 [cs.RO] https://arxiv.org/abs/2508.07917

[6] Jingwei Ma, Vivek Jayaram, Brian Curless, Ira Kemelmacher-Shlizerman, and Steven M. Seitz. 2025. UltraZoom: Generating Gigapixel Images from Regular Photos. arXiv:2506.13756 [cs.CV] https://arxiv.org/abs/2506.13756

[7] Nataniel Ruiz, Yuanzhen Li, Varun Jampani, Yael Pritch, Michael Rubinstein, and Kfir Aberman. 2023. DreamBooth: Fine Tuning Text-to-Image Diffusion Models for Subject-Driven Generation. arXiv:2208.12242 [cs.CV] https://arxiv.org/abs/2208.12242

[8] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2024. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. arXiv:2311.03285 [cs.LG] https://arxiv.org/abs/2311.03285

[9] Oriane Siméoni, Huy V. Vo, Maximilian Seitzer, Federico Baldassarre, Maxime Oquab, Cijo Jose, Vasil Khalidov, Marc Szafraniec, Seungeun Yi, Michaël Ramamonjisoa, Francisco Massa, Daniel Haziza, Luca Wehrstedt, Jianyuan Wang, Timothée Darcet, Théo Moutakanni, Leonel Sentana, Claire Roberts, Andrea Vedaldi, Jamie Tolan, John Brandt, Camille Couprie, Julien Mairal, Hervé Jégou, Patrick Labatut, and Piotr Bojanowski. 2025. DINOv3. arXiv:2508.10104 [cs.CV] https://arxiv.org/abs/2508.10104

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762