PUNICA: MULTI-TENANT LORA SERVING

Lequn Chen [*1] **Zihao Ye** [*1] **Yongji Wu** [2] **Danyang Zhuo** [2] **Luis Ceze** [1] **Arvind Krishnamurthy** [1]

ABSTRACT

Low-rank adaptation (LoRA) has become an important and popular method to adapt pre-trained models to specific domains. We present Punica, a system to serve multiple LoRA models in a shared GPU cluster. Punica contains a new CUDA kernel design that allows batching of GPU operations for different LoRA models. This allows a GPU to hold only a single copy of the underlying pre-trained model when serving multiple, different LoRA models, significantly enhancing GPU efficiency in terms of both memory and computation. Our scheduler consolidates multi-tenant LoRA serving workloads in a shared GPU cluster. With a fixed-sized GPU cluster, our evaluations show that Punica achieves 12x higher throughput in serving multiple LoRA models compared to state-of-the-art LLM serving systems while only adding 2ms latency per token. Punica is open source at `https://github.com/punica-ai/punica`.

1 INTRODUCTION

Low-rank adaptation (LoRA) (Hu et al., 2022) is becoming increasingly popular in specializing pre-trained large language models (LLMs) to domain-specific tasks with minimal training data. LoRA retains the weights of the pre-trained model and introduces trainable rank decomposition matrices to each layer of the Transformer architecture, significantly reducing the number of trainable parameters and allowing tenants to train different LoRA models at a low cost. LoRA has been integrated into many popular fine-tuning frameworks (Mangrulkar et al., 2022). Consequently, ML providers have to serve a large number of specialized LoRA models simultaneously for their tenants' needs.

Simply serving LoRA models as if they were independently trained from scratch wastes GPU resources. Assuming we need k GPUs to serve each LoRA model, serving n different LoRA models would seemingly require $k \times n$ GPUs. This straightforward approach overlooks the potential weight correlations among these LoRA models, given they originate from the same pre-trained models.

We believe an efficient system to serve multiple, different LoRA models needs to follow three design guidelines. (G1) GPUs are expensive and scarce resources, so we need to consolidate multi-tenant LoRA serving workloads to a small number of GPUs, increasing overall GPU utilization. (G2) As prior works have already noticed (Yu et al., 2022), batching is one of the, if not the most, effective approaches to consolidate ML workloads to improve performance and GPU utilization. However, batching only works when requests are for the exact same model. We thus need to enable batching for different LoRA models. (G3) The decode stage is the predominant factor in the cost of model serving. We thus only need to focus on the decode stage performance. Other aspects of the model serving are less important, and we can apply straightforward techniques, e.g., on-demand loading of LoRA model weights.

Based on these three guidelines, we design and implement Punica, a multi-tenant serving framework for LoRA models on a shared GPU cluster. One key novelty is the design of a new CUDA kernel, **Segmented Gather Matrix-Vector Multiplication** (SGMV). SGMV allows batching GPU operations for the concurrent execution of multiple, different LoRA models. With SGMV, a GPU only needs to store a single copy of the pre-trained model in memory, significantly improving GPU efficiency in terms of both memory and computation. We pair this new CUDA kernel with a series of state-of-the-art system optimization techniques.

SGMV allows batching requests from different LoRA models, and *surprisingly*, we observe negligible performance differences between batching the same LoRA models and batching different LoRA models. At the same time, the on-demand loading of LoRA models has only millisecond-level latency. This gives Punica the flexibility to consolidate user requests to a small set of GPUs without being constrained by what LoRA models are already running on the GPUs.

Punica thus schedules multi-tenant workloads in the following two ways. For a new request, Punica routes the request to a small set of active GPUs, ensuring that they reach their full capacity. Only when the existing GPUs are fully utilized, Punica will allocate additional GPU resources. For existing requests, Punica periodically migrates them for

[*]Equal contribution [1]University of Washington [2]Duke University. Correspondence to: Lequn Chen <lqchen@cs.washington.edu>.

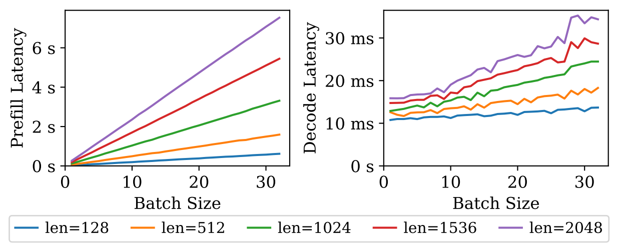

Figure 1. Batching effects in Prefill stage and in Decode stage

consolidation. This allows freeing up GPU resources that are allocated to Punica.

We evaluate LoRA models that are adapted from Llama2 7B, 13B, and 70B models (Touvron et al., 2023) on NVIDIA A100 GPU clusters. Given the same amount of GPU resources, Punica achieves 12x higher throughput compared to state-of-the-art LLM serving systems while only adding 2ms latency per token.

This paper makes the following contributions:

- We identify the opportunity of batch processing requests of multiple, different LoRA models.
- We design and implement an efficient CUDA kernel for running multiple LoRA models concurrently.
- We develop new scheduling mechanisms to consolidate multi-tenant LoRA workloads.

2 Background

We first present the text generation process for transformer models. We then describe Low-Rank Adaptation (LoRA) of transformer models.

2.1 Transformer and Text Generation

Transformer-based LLMs operate on a sequence of tokens. A token is roughly ¾ of an English word. An LLM's operation consists of two stages. The *prefill* stage accepts a user prompt and generates a subsequent token and a Key-Value cache (KvCache). The *decode* stage accepts a token and the KvCache, and it then generates one more token and appends a column in the KvCache. The decode stage is an iterative process. The generated token then becomes the input for the next step. This process ends when the end-of-sequence token is generated.

A transformer block contains a self-attention layer and a multilayer perceptron (MLP). Let us assume that the length of the prompt is s and the attention head dimension is d. For the prefill stage, the computation of the self-attention layer is $(s,d)\times(d,s)\times(s,d)$, and the MLP computation is $(s,h)\times(h,h)$. For a decode step, assuming s represents the past sequence length, the computation of the self-attention layer is $(1,d)\times(d,s+1)\times(s+1,d)$ and the MLP computation is $(1,h)\times(h,h)$. The decode stage has low GPU utilization because the input is a single vector.

Figure 1 shows the latency for the prefill stage and the decode stage for different batch sizes. The computation capability of the GPU is fully utilized during the prefill stage. Prefill latency is proportional to batch size. However, this is not the case for the decode stage. Increasing the batch size from 1 to 32, the decode step latency increases from 11ms to 13ms for short sequences, and from 17ms to 34ms for longer sequences. This means that batching can improve GPU utilization significantly for the decode stage. Orca (Yu et al., 2022) leveraged this opportunity to build an efficient LLM serving system. This type of batching is especially important because the decode stage predominately determines the serving latency for long output length responses.

2.2 Low-Rank Adaptation (LoRA)

Fine-tuning allows a pre-trained model to adapt to a new domain or a new task or be improved with new training data. However, because LLMs are large, fine-tuning all the model parameters is resource-intensive.

Low-Rank Adaptation (LoRA) (Hu et al., 2022) significantly reduces the number of parameters needed to be trained during fine-tuning. The key observation is that the weight difference between the pre-trained model and the model after fine-tuning has a low rank. This weight difference can thus be represented as the product of two small and dense matrices. LoRA fine-tuning then becomes similar to training a small, dense neural network. Formally, let's consider the weights of the pre-trained model to be $W\in\mathbb{R}^{h_1\times h_2}$. LoRA fine-tuning trains two matrices $A\in\mathbb{R}^{h_1\times r}$ and $B\in\mathbb{R}^{r\times h_2}$, where r is the LoRA Rank. $W+AB$ is the new weight for the fine-tuned model. LoRA rank is usually much smaller than the original dimension (e.g., 16 instead of 4096). In addition to fast fine-tuning, LoRA has very low storage and memory overheads. Each fine-tuned model only adds 0.1% to 1% of the model weight. LoRA is usually applied to all dense projections in the transformer layer (Dettmers et al., 2023), including the Query-Key-Value-Output projections in the attention mechanism and the MLP. Note that the self-attention operation itself does not contain any weight.

How to serve multi-tenant LoRA models efficiently on a shared GPU cluster? LoRA provides an efficient algorithm to fine-tune LLMs. Now the question is: how to serve those LoRA models efficiently? One approach is to regard each LoRA model as an independent model and use traditional LLM serving systems (e.g., vLLM). However, this neglects the weight sharing among different LoRA models

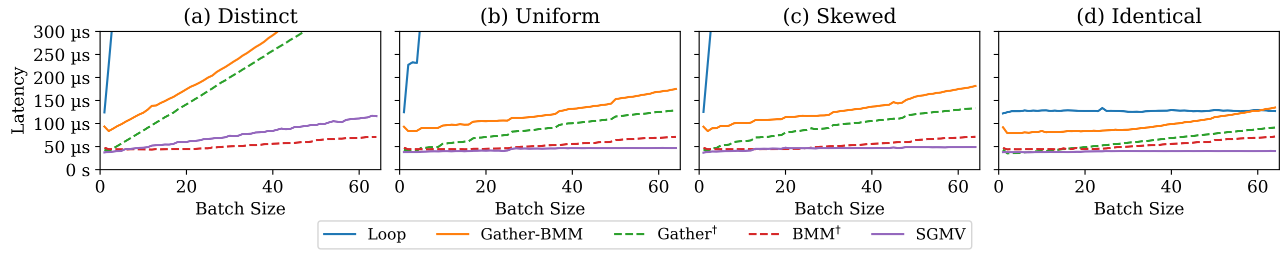

Figure 8. Microbenchmark for LoRA operator implementations. †Gather and BMM are measured separately for reference.

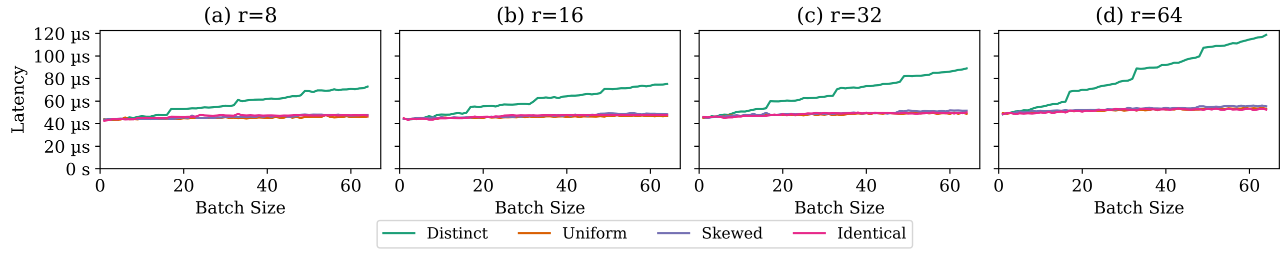

Figure 9. Microbenchmark for LoRA operator on various LoRA rank.

Overall, SGMV significantly outperforms baseline implementations regardless of workloads.

We also run the microbenchmark of different LoRA ranks on Testbed #1. Figure 9 shows the latency for LoRA rank 8, 16, 32, and 64. In the **Distinct** case, the latency gradually increases. The latency of a single request batch is around 42µs for all four ranks, while batch size 64 goes up to 72µs, 75µs, 89µs, and 118µs, respectively. When the workload exists weight sharing (**Uniform**, **Skewed**, and **Identical**), the latency remains almost the same across batch size 1 to 64, at around 42µs to 45µs.

Transformer layer benchmark Next, we evaluate the transformer layer performance after incorporating the LoRA operator. Since the LLM is roughly a stack of transformer layers, the layer performance determines the overall model performance. We run the layer benchmark on Testbed #1 based on the 7B and 13B model configurations and sequence lengths of 512 and 2048. Figure 10 plots the layer latency. When the sequence length is shorter, the batching effect is stronger. The latency only increases by 72% when batch size increases from 1 to 32 when the sequence length is 512. When the sequence is longer, self-attention takes longer time, which reduces the layer-wise batching effect.

In contrast to the kernel microbenchmark, notice that the layer latency is roughly the same across different workloads. This is because the computation time for the LoRA add-on is small compared to the backbone dense projection and the self-attention. *This LoRA-model-agnostic performance property enables us to schedule different LoRA models as if one model.* Our scheduling algorithm can then focus on the overall throughput instead of individual LoRA model placement, which is exactly how we design Punica.

7.2 Text generation

Next, we study the text generation performance of Punica and baseline systems.

Serving 7B and 13B models on a single GPU We evaluate text generation using Punica and baseline systems on a single GPU on Testbed #1. The single-GPU performance serves as the base case for cluster-wide deployment. We generate 1000 requests (generating around 101k tokens) and restrict each system to batch in a first-come-first-serve manner. The max batch size is set to 32 for all systems. Punica can batch across different LoRA models, and baseline systems can only batch requests for the same LoRA models.

Figure 11 (a) and (b) show the results on the 7B model and the 13B model, respectively. Punica consistently delivers high throughput regardless of workloads. Punica achieves 1044 tok/s and 693 tok/s on the 7B and the 13B models, respectively. Although most baselines can achieve relatively high throughput in the **Identical** case, their performance deteriorates when there are multiple LoRA models.

In the **Distinct** case, all baseline systems run with a batch size of 1, and thus, the throughput is low. In the **Uniform** and the **Skewed** cases, most batches for the baseline systems have extremely small batch sizes (1–3), which explains the