# 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>.

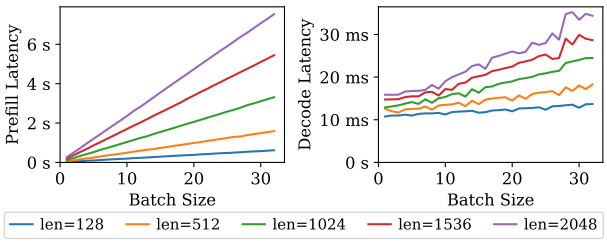


*Figure 1.* Batching effects in Prefill stage and in Decode stage

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

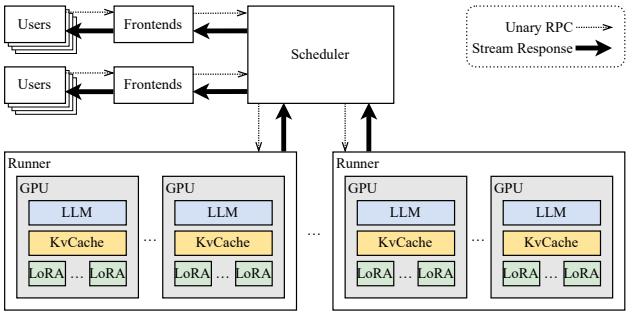


*Figure 2.* The system architecture of Punica.

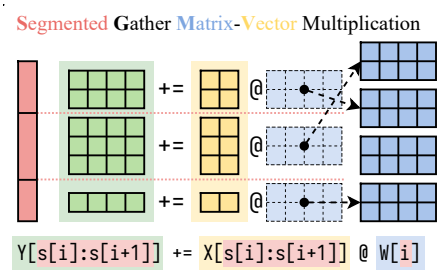


*Figure 3.* Semantics of SGMV.

that can be used to significantly improve GPU efficiency. Further, if we treat each LoRA model as an independent one, model loading time can be a substantial performance bottleneck when bootstrapping model serving. Even if we share the backbone model across the LoRA models, it remains a question as to how to batch compute the LoRA addon efficiently.

## 3 Punica Overview

We design Punica as a multi-tenant system that manages a cluster of GPUs to serve multiple LoRA models with shared pre-trained backbone models. Figure 2 shows the system architecture of Punica. Like other model serving systems, Punica has frontend servers that expose RESTful API to end-users and forward users' serving requests to the Punica scheduler. A user request contains the identifier of the LoRA model and a prompt. The scheduler dispatches requests to the GPUs. Each GPU server starts a runner, which communicates with the scheduler and controls the execution of all the GPUs. As GPUs generate new tokens, new tokens are streamed from the runners to the scheduler, to the frontends, and finally to the end-users.

In Punica, each GPU loads the backbone pre-trained large language model. A large fraction of GPU memory is reserved for KvCache. Only the LoRA components of models are swapped in from remote storage when needed. Note that this design allows fast cold-start for model serving. Because the pre-trained model is already loaded into the GPU memory, Punica only needs to load matrices A and B for a new LoRA model.

Punica needs to address two key research challenges. The first challenge is how to run multiple LoRA models efficiently on a GPU. Because requests have to be served by different LoRA models, each request has to go through a different GPU computation. We use the existing matrix multiplication for the backbone computation. And we present a new CUDA kernel for adding the LoRA addons to the backbone computation in a batched manner. We call this kernel *Segmented Gather Matrix-Vector Multiplication* (SGMV). SGMV parallelizes the feature-weight multiplication of different requests in the batch and groups requests corresponding to the same LoRA model to increase the operational intensity of the kernel and use GPU Tensor Cores units for acceleration.

The second challenge is how to design an efficient system on top of SGMV for multi-tenant LoRA model serving. Our goal here is to consolidate multi-tenant workloads to the smallest set of GPUs possible, occupying the least amount of GPU resources. Punica schedules user requests to active GPUs, which already serve or train LoRA models. This is feasible in Punica, because with SGMV adding the batch size, even if for different LoRA models, improves the GPU utilization. For old requests, Punica migrates them periodically to consolidate the workloads, thereby freeing up GPU resources.

Next, we describe the details of Punica's CUDA kernel and other design details of Punica.

## 4 Segmented Gather Matrix-Vector Multiplication

When a LoRA model has multiple inputs in the batch, we can further batch them together. We group inputs to the same LoRA model consecutively. Denote $n$ as the number of LoRA models in a batch. Denote sequence $s_i$ as the last element index for $i$-th model within the batch. In particular, $s_0 = 0$ and $s_n$ is the batch size. Input $\{\vec{x_i} \mid i \in [1, s_n]\}$ is then partitioned as $\{\{\vec{x_j} \mid j \in (s_{n-1}, s_n]\} \mid i \in [1, n]\}$. The dense projection output can then be written as:

$$\begin{pmatrix} \vdots \\ \vec{y}_{s_1} \\ \vdots \\ \vec{y}_{s_{n-1}+1} \\ \vdots \\ \vec{y}_{s_n} \end{pmatrix} := \begin{pmatrix} \vdots \\ \vec{x}_{s_1} \\ \vdots \\ \vec{x}_{s_{n-1}+1} \\ \vdots \\ \vec{x}_{s_n} \end{pmatrix} W + \begin{pmatrix} \vdots \\ \vec{x}_{s_1} \\ \vdots \\ \vec{x}_{s_{n-1}+1} \\ \vdots \\ \vec{x}_{s_n} \end{pmatrix} A_1 B_1 \quad \cdots \quad \begin{pmatrix} \vdots \\ \vec{x}_{s_{n-1}+1} \\ \vdots \\ \vec{x}_{s_n} \end{pmatrix} A_n B_n$$

The left-hand-side multiplication is the computation for the backbone model, which is batched through regular GEMM.

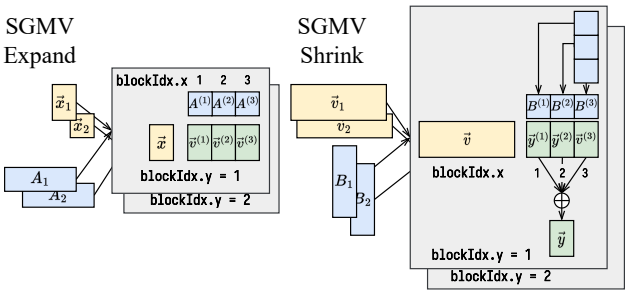


*Figure 4.* Scheduling of SGMV expand/shrink kernels

We need a fast kernel to compute the right-hand-side LoRA addon. Note that operator $\vec{y} \mathrel{+}= \vec{x}AB$ can be separated as two launches of the same kernel: Initialize $\vec{v} := \vec{0}$. Then we run $\vec{v} \mathrel{+}= \vec{x}A$ and follow by $\vec{y} \mathrel{+}= \vec{v}B$.

We name this operator SGMV, Segmented Gather Matrix-Vector multiplication. Figure 3 illustrates the semantics of SGMV.

**CUDA Kernel Schedule** We classify SGMV operator into two categories, SGMV-shrink and SGMV-expand, based on their input and output feature dimensions. The first operator in the LoRA module: $\vec{v} = \vec{x}A$ is SGMV-shrink because it shrinks a high-dimensional input feature to low-rank output. The second operator $\vec{y} = \vec{v}B$ is SGMV-expand as it expands the low-rank input feature to a high-dimensional output feature.

Figure 4 shows how we schedule the SGMV kernel in these two cases: For both kernels, we bind the LoRA index to BLOCKIDX.Y in CUDA. Then, the computation on each BLOCKIDX.Y is a matrix multiplication between features and a specific LoRA weight. We designed different schedules for matrix multiplication under expand and shrink settings: for the expand kernel, we split $A$ on the output feature dimension $A = \begin{bmatrix} A^{(1)} & \cdots & A^{(n)} \end{bmatrix}$ and dispatch different $\vec{v}^{(i)} = \vec{x}A^{(i)}$ to different threadblocks in GPU, and the concatenation $\vec{v}^{(i)}$ on different threadblocks forms the final result $\vec{v} = \begin{bmatrix} \vec{v}^{(1)} & \cdots & \vec{v}^{(n)} \end{bmatrix}$; for the shrink kernel, the output dimension is too thin and we adopt the Split-K strategy (Thakkar et al., 2023) to increase parallelism: we split $B$ on the input feature dimension $B = \begin{bmatrix} B^{(1)} \\ \cdots \\ B^{(k)} \end{bmatrix}$. We dispatch different $\vec{y}^{(i)} = \vec{v}B^{(i)}$ to different threadblocks in GPU, after the partial sum $\{\vec{y}^{(i)}\}$ computation on all threadblocks are finished, we perform a grid synchronization followed by a cross threadblock reduction $\vec{y} = \sum_{i=1}^{k} \vec{y}^{(i)}$ to aggregate the partial results. We use GPU Tensor Cores to accelerate matrix multiplication for both kernels.

In the case that each request has a distinct LoRA index, the computation corresponding to each LoRA index degrades to matrix-vector multiplication, which is totally IO-bound. We designed a specific schedule for this case that maximizes memory bandwidth utilization and does not use Tensor Cores because of the low operational intensity of the operator.

## 5 PUNICA IN DETAIL

Punica schedules new user requests at a per-request level and migrates old requests between GPUs at a per-iteration level. The scheduler adds requests to a GPU or cancels a working request from a GPU. Each GPU batches all requests in its working set for LLM invocation. GPU runs the Prefill steps and Decode steps continuously. When a request reaches the stopping condition (end-of-sequence token or length limit), the GPU removes the request from the batch and notifies the scheduler about the stopping.

We run batch requests of prefill and decode stages in a single model invocation. To minimize latency penalty, we limit the prefill batch size to 1 for each batch. The single prefill and the batch of decodes invoke two separate CUDA kernels for the self-attention operation. All other operations, including dense projection and LoRA addon, treat all tokens in the prefill stage and decode stage as a single batch input. In this way, we increase the batch efficiency of dense projection and LoRA addon.

### 5.1 Scheduling new requests

Punica scheduler has a global view of the state of all the GPUs. In particular, for each GPU, Punica maintains the working set of requests, which is the batch input of LLM invocation. As new requests are added to the working set and as the decode steps unfold, the KvCache consumes more and more GPU memory. Therefore, Punica also continuously tracks the memory space available for KvCache on each GPU.

Punica schedules a new request to the GPU that currently has the largest working set of requests (i.e., the LLM invocation batch size) while satisfying the following constraints: (1) It has not yet reached the max batch size limit. (2) It has enough memory for the new request's KvCache. When there are multiple candidates, the one that has the highest GPU UUID gets the new request. When all GPUs are fully occupied (i.e., have reached the maximum batch size or have insufficient memory), the request is queued. When some GPUs become available in the future, queued requests are scheduled in a first-come-first-serve (FCFS) manner.

The max batch size limit balances the cluster throughput and the per-token latency. Oversized batches greatly slow down latency while providing marginal throughput gains. We profile A100 GPUs and decide to set the maximum batch

size to 32.

The GPU selection logic emphasizes cluster throughput within the latency sweet spot. Our scheduling approach has the following attributes: a busy GPU is likely to stay busy as more requests will be assigned to it, a lightly loaded GPU is likely to lower its load as requests terminate, and an idle GPU is likely to stay idle. As a consequence, our scheduler maintains peak throughput and consolidates GPU usage based on the current overall system load. This allows easier decisions to scale up/down the GPU cluster. In a cloud setting where Punica can allocate and deallocate GPU servers, we perform the following cluster allocations: (1) If no lightly loaded GPU exists in the cluster, Punica should request more GPUs. (2) Punica can return the GPU resources for GPU servers with no load.

### 5.2 On-demand model loading

Weight sharing between the LoRA model and the underlying pre-trained model makes model loading fast. The size of the LoRA model (which is matrices $A$ and $B$ in §2.2) is only 1% of the underlying pre-trained model.

Loading a LoRA model from the main memory to the GPU memory is merely an asynchronous host-to-device memory copy. The latency is bounded by the PCIe bandwidth. On PCIe Gen4 x16, it takes around 50µs to load a layer and 2ms to load the entire model. Since the memory copy and the GPU computation can overlap, it is feasible to implement sophisticated layer-by-layer or even matrix-by-matrix loading to minimize the model loading delay.

However, notice that each decode step takes around 30ms to complete, and each request might need thousands of decode steps. We opt to use a simpler yet equivalently efficient method. When a request is newly added to a GPU, if its LoRA model is not already loaded, we issue an asynchronous memory copy to load the LoRA weight, and let the GPU continue running other inputs in the batch. By the end of the model execution, the weight already finished loading. Then, the new request is able to join the batch naturally.

### 5.3 Request migration

As each request generates more tokens, their KvCache occupies more GPU memory. When a GPU runs out of space for KvCache, it migrates some requests to other GPUs. The request migration consists of two steps — evict and add. The scheduler evicts the newest request from the GPU. This preserves the FCFS semantics. The scheduling for the evicted request is the same as adding a new request.

Punica scheduler supports canceling requests. Cancellation is straightforward: remove the request from both the GPU and the scheduler states. A typical scenario for cancellation is user disconnection. More importantly, request cancellation as a scheduling primitive enables request migration.

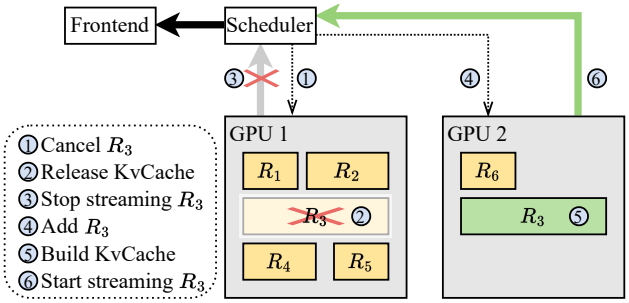


*Figure 5.* Request migration procedure for Request $R_3$.

Figure 5 shows the workflow to migrate a request, $R_3$, from GPU 1 to GPU 2. The scheduler first sends the cancellation of the request to GPU 1. After GPU 1 finishes running the previous batch, it picks up the cancelation and releases KvCache. GPU 1 also omits the $R_3$'s new token generated in the previous batch. Immediately after sending the cancellation to GPU 1, the scheduler adds $R_3$ to GPU 2. GPU 2 runs a prefill step on $R_3$'s original prompt plus all previously generated tokens. This reinstablishes the $R_3$'s KvCache on GPU 2. GPU 2 then starts to stream $R_3$'s new tokens to the scheduler.

We opt for recomputation instead of moving the KvCache for its simplicity. PagedAttention (Kwon et al., 2023) has shown that recomputation's latency is equal to or better than moving the KvCache in most cases, which agrees with our observation.

### 5.4 Memory layout for KvCache

Punica uses a separable KvCache layout, which is important for text generation batching throughput. The HuggingFace Transformers library's KvCache layout consists of complicated nested lists of tensors, which can conceptually be viewed as the following shape:

$$[L, 2, B, N, S, D]$$

where $L$ is the number of layers, 2 is for Key and Value projection, $B$ is batch size, $N$ is the number of heads, $S$ is the sequence length, and $D$ is the head dimension. In each decode step, HuggingFace Transformers concatenates one tensor along the sequence length dimension. The concatenation is inefficient as it needs to read the whole KvCache and write a new copy, whereas the new tensor is only $1/S$ of the KvCache.

The bigger problem with the HuggingFace Transformer's approach is that the batching dimension is not the outermost dimension, which means that requests in the batch are hard to separate in the KvCache. Under this restriction, requests

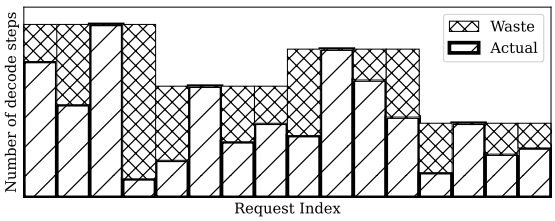


*Figure 6.* Inseparable KvCache adds wasted decode steps.

that enter the batch together need to remain together during all decode steps *until all requests meet their own stopping condition*.

Figure 6 is an illustrative figure that explains the problem. In the figure, consecutive 4 requests are batched together. The striped bar represents the number of decode steps that each request actually needs to reach its stopping condition. Due to inseparable KvCache, shorter requests in the batch run additional decode steps, which is essentially wasted computation. FasterTransformer (Hsueh, 2021) and DeepSpeed (Aminabadi et al., 2022) also suffer from similar problems.

Instead, our KvCache layout is

$$[\textstyle\sum_i \lceil S_i/P \rceil, L, 2, N, P, D]$$

where $S_i$ is the length of the sequence $i$ and $P$ is the page size. We use paged KvCache (Kwon et al., 2023) to minimize memory fragmentation. We put the batching dimension on the outmost to enable continuous batching.

## 6 Implementation

Punica implementations consist of two parts: a Python library on top of PyTorch that runs large language models on a single GPU and other system components to support model serving across a GPU cluster.

**Python Library** We expose our CUDA kernels as a PyTorch Extension using PyBind11. Llama model implementation is adapted from the HuggingFace Transformers library. We use FlashInfer (Ye, 2023) open source project for fast and memory-efficient computation of self-attention. Besides fusing the computation of $\mathrm{softmax}(QK^T)V$ like FlashAttention (Dao et al., 2022) does, FlashInfer further supports batch decoding without padding. Similar to PagedAttention (Kwon et al., 2023), FlashInfer supports paged KvCache to minimize GPU memory fragmentation due to KvCache. We also fuse LayerNorm, which reduces latency from 110µs to 4µs.

We mix new requests in the Prefill stage and existing requests in the Decode stage in one batch together. This way, dense projections and LoRA can benefit from a bigger batch size. For batching, we concatenate all inputs along the sequence length dimension. We always put Prefill requests at the beginning and Decode requests at the latter part. We then pass a `BatchLen` struct to distinguish different requests. `BatchLen` contains a list of indices indicating the starting index of each Prefill request. It also contains a number indicating the number of Decode requests. In the self-attention layer, we pass the indices and leading input states to the BatchPrefill kernel, and we pass the trailing input states to the BatchDecode kernel. Within a batch, we further organize the batch input order such that requests that share the same LoRA model are consecutive. The tail of Prefill requests and the head of Decode requests can share a LoRA model if possible. We then generate the segment indices for the SGMV kernel. Before each batched model invocation, we concatenate batch inputs and construct `BatchLen` and SGMV segment indices. Both `BatchLen` and SGMV segment indices remain constant for the entire model invocation. This design avoids recomputation ($L$ times for `BatchLen` and $7L$ times for SGMV segment indices, where $L$ is the number of layers).

**Other system components** We write our scheduler, frontend, and runner in Rust. Unary RPC and streaming text chunks are both implemented via web socket. I/Os are handled asynchronously. Runner spawns a Python subprocess for each GPU. The subprocess is a thin warper around our PyTorch library. The Runner main process communicates with the subprocesses using pipes.

## 7 Evaluation

We evaluate Punica on two testbeds. Testbed #1 is a single server with one NVIDIA A100 80GB GPU. Testbed #2 consists of two NVIDIA HGX A100 40GB servers with 8 GPUs on each server. Testbed #1 contains a GPU with large GPU memory, which allows us to study the LoRA batching effect. Testbed #2 is equipped with fast NvSwitch technology for us to study tensor parallelism and evaluate cluster deployment. We use the Llama-2 (Touvron et al., 2023) model with 7B, 13B, and 70B parameters. For all experiments, we use 16 as the LoRA rank. LoRA is applied to all dense projections. We use random weights for LoRA models as the weight does not affect latency performance.

**Workloads** Prompt lengths and response lengths are key workload characteristics for LLM serving. We use the prompt and response length distributed from ShareGPT (ShareGPT, 2023), which consists of user-bot conversations from Internet users. We consider four types of request distribution among LoRA models. (1) **Distinct**: each request is for a distinct LoRA model. (2) **Uniform**: all LoRA models are equally popular. Given $n$ requests, we use $\lceil \sqrt{n} \rceil$ models. (3) **Skewed**: model popularity follows Zipf-$\alpha$ distribution. The number of requests to the $i$-th most popular model is $\alpha$ times that of the $i+1$-th's. In our exper-

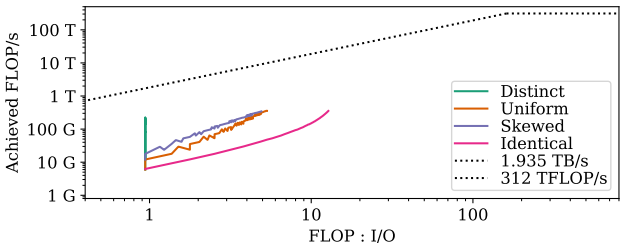


*Figure 7.* Roofline plot of the SGMV kernel.

iments, we choose $\alpha$ to be 1.5. (4) **Identical:** all requests are for the same LoRA model.

**Baselines** Since there is no well-known multi-LoRA serving system, we compare Punica against a variety of popular LLM backbone serving systems. We allow various degrees of relaxation that are in favor of baseline systems. We use HuggingFace PEFT (Mangrulkar et al., 2022) library to add LoRA weights to HuggingFace Transformers (Wolf et al., 2020) library and DeepSpeed (Aminabadi et al., 2022). We run backbone-only for FasterTransformer (Hsueh, 2021) and vLLM[1] (Kwon et al., 2023) since these two systems do not support LoRA models. We omit the model switching costs for baseline systems.

### 7.1 Microbenchmarks

We use analysis and testbed evaluations to benchmark SGMV and distill the -performance implications for the LoRA operator and a single transformer layer.

**Roofline analysis for SGMV** First, we use the roofline model (Williams et al., 2009) to understand the performance of our SGMV kernel. The number of floating point operations (FLOP) and the number of memory I/O bytes of SGMV are calculated as:

$$\begin{aligned} \text{FLOP} &= s_n \times h_i \times h_o \times 2 \\ \text{I/O} &= [s_n \times (h_i + h_o) + n \times h_1 \times h_2] \times 2 \end{aligned}$$

where $n$ is the number of LoRA models, $s$ is the segment indices, $s_n$ is the total number of inputs, and $h_i$ and $h_o$ are the input and output dimensions of the SGMV weight matrix. The factor 2 in FLOP comes from multiply-add operations for matrix multiplication. The factor 2 in I/O comes from the byte size of 16-bit floating point data type. We use $h_i = 16$, $h_o = 4096$ for this case study. We measure the latency of batch size 1 to 64 under the four different popularity distributions on Testbed #1.

Figure 7 shows the roofline model plot. The x-axis in the roofline model is arithmetic intensity, which is defined as the ratio of FLOP to I/O. The y-axis is the achieved throughput in terms of FLOP per second, calculated using measured latency. The diagonal dotted line and the top dotted line represent the memory bandwidth and peak FP16 performance of the NVIDIA A100 GPU, respectively.

In the **Distinct** case, the arithmetic intensity does not change because FLOP and I/O grow at the same rate. Since each input only utilizes a small amount of GPU compute units, increasing the batch size increases performance. In the **Identical** case, the line goes up diagonally following the slope of memory bandwidth, which means that SGMV is bounded by memory bandwidth. The **Uniform** case and the **Skewed** case sit in between, as a combination of both effects, the increasing degree of parallelism and the increasing arithmetic intensity.

Analysis of performance

**LoRA operator microbenchmark** We implement the batched LoRA operator as two SGMV kernel launches. We compare our SGMV-based implementation against two PyTorch-based baseline implementations. One is a for-loop over each LoRA model. Another is Gather-BMM. In the gather step, we stack the weight matrices that each input needs into a single matrix. Then, we use `torch.bmm()` to perform a batched matrix multiplication on the input and the stacked matrix. Similar to SGMV, Gather-BMM launches Gather twice and BMM twice. Note that Gather-BMM uses much more I/O than SGMV. Gather reads in $n \times h_i \times h_o$ elements and writes to $s_n \times h_i \times h_o$. Then, BMM needs to read in $s_n \times h_i \times h_o$ weight elements that Gather just wrote. In combination, Gather-BMM incurs $s_n \times h_i \times h_o \times 2$ more elements memory I/O than SGMV.

Figure 8 shows the latency comparison of the three implementations across four workloads on Testbed #1. Gather and BMM are also measured separately for reference. Since BMM is data-independent, its latency is consistent across four workloads.

Our benchmark results match our analysis very well. In the **Distinct** case, Loop behaves terribly because it runs multiple rounds of batch size 1. Gather-BMM latency increases fast due to the slowdown of Gather. SGMV latency increases gradually as well, from 37µs to 116µs, because batching does not change arithmetic intensity. The **Uniform** case and the **Skewed** case are similar to the **Distinct** case. Gather-BMM performs slightly better than the **Distinct** case since there are fewer matrices to read. SGMV latency increases only marginally, from 37µs to 46µs, as a combination of both effects: increasing degree of parallelism and increasing arithmetic intensity. In the **Identical** case, all implementations have the same semantics: BMM. We can, therefore, infer that SGMV implements BMM more efficiently than `torch.bmm()` in the case of LoRA. SGMV latency remains almost constant, from 37µs to 40µs.

[1]With commit `928de46`

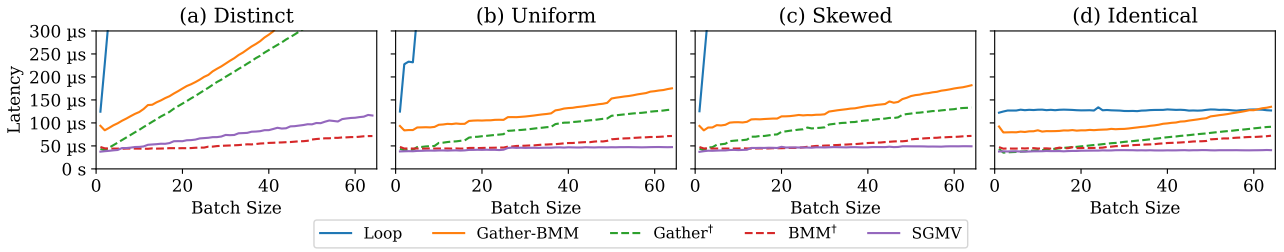


*Figure 8.* Microbenchmark for LoRA operator implementations. †Gather and BMM are measured separately for reference.

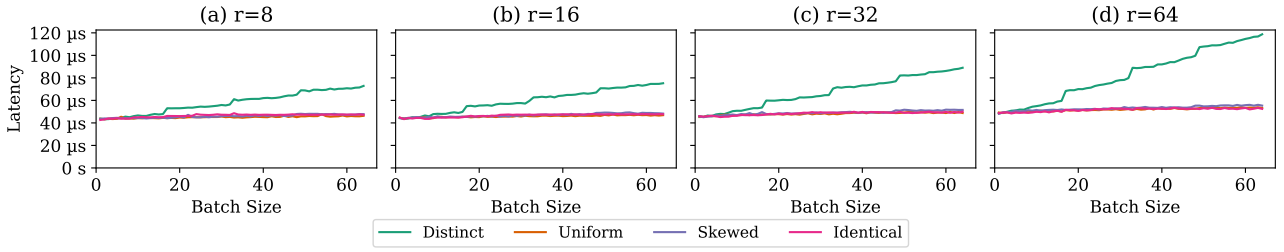


*Figure 9.* Microbenchmark for LoRA operator on various LoRA rank.

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

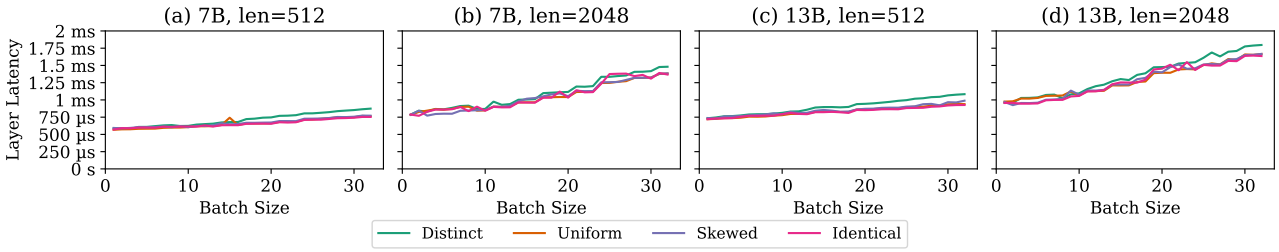


*Figure 10.* Transformer Layer Benchmark.

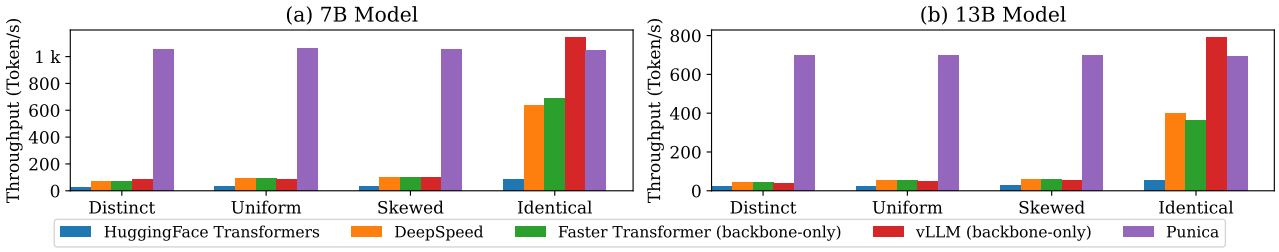


*Figure 11.* Single GPU text generation comparison

low performance. Punica is able to batch different LoRA models in one batch and, therefore, can run with a batch size of 32 consistently.

With only one LoRA model, all systems can run with a batch size of 32. Thus, all except for the HuggingFace Transformer can deliver high throughput. HuggingFace Transformer's low performance is due to its lack of critical CUDA kernel optimizations, including FlashAttention (Dao et al., 2022). In the **Identical** case, both vLLM and Punica outperform other systems because the two systems' Kv-Cache layout allows continuous batching. In contrast, other systems have to wait for the longest sequence in the batch to finish. vLLM's throughput is slightly higher than Punica (at 1140 tok/s and 789 tok/s, respectively) because we run vLLM backbone-only.

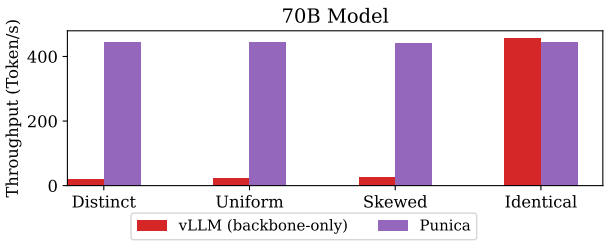


*Figure 12.* 70B model text generation comparison.

**Serving 70B models with tensor parallelism** We run the 70B model on Punica with Megatron's tensor parallel scheme (Shoeybi et al., 2019; Narayanan et al., 2021) on 8 GPUs in Testbed #2. We compare Punica and vLLM. vLLM also uses the same Megatron's tensor parallel scheme.

Figure 12 shows similar trends as our results in serving 7B and 13B models. In the presence of multiple LoRA models, vLLM's throughput is around 21 to 25 tok/s, whereas when serving the backbone, vLLM can achieve 457 tok/s due to the large batch size. For the **Identical** case, Punica and vLLM achieve the same performance because their parallel schemes are the same. However, Punica can consistently deliver 441 to 446 tok/s throughput regardless of LoRA popularity distribution, significantly outperforming vLLM for serving multiple LoRA models.

### 7.3 Cluster deployment

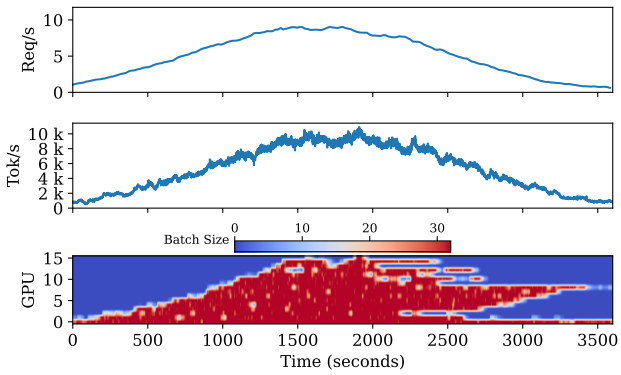


*Figure 13.* Cluster deployment.

We evaluate Punica on 16 GPUs in Testbed #2. The load varies as follows: In the macro view, the request rate of the

workload gradually increases and then gradually decreases. In the micro view, gaps between request arrival time follow an exponential distribution, and the arrival process follows a Poisson distribution. LoRA model popularity follows Zipf-1.5 distribution (same as our **Skewed** workload). The duration of the experiment is one hour. The model size is 7B in this experiment.

Punica is able to consolidate GPU usage while delivering high throughput. Figure 13's upper panel shows the request rate over time. The middle panel shows text generation throughput in terms of tokens per second. The lower panel shows the batch size of each GPU across time. GPUs usually run with the maximum batch size when they are not idle because our schedule algorithm prioritizes large batch sizes. Occasionally, a GPU runs with a smaller batch size because it runs out of KvCache space and migrates out a few requests to other GPUs. When a GPU becomes idle (batch size = 0), it is likely that it stays idle, which can then be released to the cloud provider if necessary.

## 8 Related Work

**LLM inference optimization.** A series of recent work has focused on optimizing LLM inference. Orca (Yu et al., 2022) proposes batching transformer-based text generation by splitting concatenated batch input at the self-attention operation. vLLM (Kwon et al., 2023) further reduces the memory fragmentation of KvCache by borrowing the idea of virtual pages in operating systems. FlashAttention (Dao et al., 2022) provides an optimized implementation of self-attention operation by reducing data movement via block-wise computation. Punica already integrates them. On the other hand, FlexGen (Sheng et al., 2023) designed an efficient swapping schedule to maximize throughput on a single GPU while sacrificing latency. Speculative Decoding (Leviathan et al., 2023; Miao et al., 2023; Cai et al., 2023) increases the operational intensity of auto-regressive models by using a lightweight "draft" model to propose candidates for the next $k$ tokens and verifying these $k$ tokens in parallel with large models. Punica is orthogonal to these optimizations.

**Multi-model inference serving** A substantial body of work has been proposed for serving ML models on a GPU cluster. Clipper (Crankshaw et al., 2017) is one of the earliest systems to optimize both throughput and latency in a GPU cluster. It is followed up by a series of systems (Gujarati et al., 2020). However, they are mainly designed to serve smaller CNN models. One key difference is that serving CNN models is stateless whereas LLM serving needs to persist the KvCache. The state introduces an affinity that asks for a different system design. For example, Symphony (Chen et al., 2023) uses a non-work-conserving scheduler but Punica runs batches on a GPU back-to-back due to the KvCache affinity. Although Nexus (Shen et al., 2019) supports prefix sharing of different models, they offer limited support and optimization faced with LLMs and fine-grained sharing patterns as we witnessed in LoRA.

PetS (Zhou et al., 2022) batches requests to different adapters (e.g., Adapters (Houlsby et al., 2019), MaskBert (Zhao et al., 2020), Diff-Pruning (Guo et al., 2021), Bitfit (Zaken et al., 2022)) of a LLM on a single GPU. It allows GPU memory sharing of the pre-trained model for different downstream tasks, however, it does not enable multiple different models to run concurrently.

**Model and KvCache quantization/compression** A substantial amount of work has been proposed to reduce the memory footprint of model weights, activations and KvCache by quantization (Frantar et al., 2022; Xiao et al., 2023a; Guo et al., 2023; Lin et al., 2023; Sheng et al., 2023). Model quantization saves more headroom for KvCache, hence enabling Punica to serve requests of longer sequences without migration. In addition, KvCache quantization (Sheng et al., 2023) and compression (Liu et al., 2023b;a; Zhang et al., 2023; Xiao et al., 2023b) further reduces the memory I/O of the KvCache, through which inference latency can be reduced, as self-attention is bounded by GPU memory bandwidth (Dao et al., 2022). QLoRA (Dettmers et al., 2023) proposes to fine-tune LoRA by storing LoRA weights/gradients in high-precision formats such as fp16 while keeping the original weight in quantized formats to save memory footprint during fine-tuning. Quantization reduces self-attention latency, which makes the high efficiency of Punica's LoRA kernel even more important.

## 9 Conclusion

Low-rank adaptation (LoRA) has become an important fine-tuning 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's design is centered around a new CUDA kernel design that allows batching of GPU operations for different LoRA models. For each GPU, Punica only requires a single copy of the underlying pre-trained model for the GPU to serve multiple, different LoRA models, significantly improving GPU efficiency in terms of both memory and computation. Additionally, Punica's 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 LoRA model serving throughput compared to state-of-the-art LLM serving systems while only adding 2ms latency per token.

## REFERENCES


Aminabadi, R. Y., Rajbhandari, S., Zhang, M., Awan, A. A., Li, C., Li, D., Zheng, E., Rasley, J., Smith, S., Ruwase, O., and He, Y. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

Cai, T., Li, Y., Geng, Z., Peng, H., and Dao, T. Medusa, September 2023. URL https://github.com/FasterDecoding/Medusa.

Chen, L., Deng, W., Canumalla, A., Xin, Y., Philipose, M., and Krishnamurthy, A. Symphony: Optimized model serving using centralized orchestration, 2023.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 613–627, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: accurate post-training quantization for generative pre-trained transformers. *CoRR*, abs/2210.17323, 2022. doi: 10.48550/ARXIV.2210.17323. URL https://doi.org/10.48550/arXiv.2210.17323.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/gujarati.

Guo, C., Tang, J., Hu, W., Leng, J., Zhang, C., Yang, F., Liu, Y., Guo, M., and Zhu, Y. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–15, 2023.

Guo, D., Rush, A. M., and Kim, Y. Parameter-efficient transfer learning with diff pruning. In Zong, C., Xia, F., Li, W., and Navigli, R. (eds.), *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pp. 4884–4896. Association for Computational Linguistics, 2021. doi: 10.18653/V1/2021.ACL-LONG.378. URL https://doi.org/10.18653/v1/2021.acl-long.378.

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for NLP. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2790–2799. PMLR, 2019. URL http://proceedings.mlr.press/v97/houlsby19a.html.

Hsueh, B. Y. FasterTransformer. https://github.com/NVIDIA/FasterTransformer, 2021.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286. PMLR, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

Liu, Z., Desai, A., Liao, F., Wang, W., Xie, V., Xu, Z., Kyrillidis, A., and Shrivastava, A. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. *CoRR*, abs/2305.17118, 2023a. doi: 10.48550/ARXIV.2305.17118. URL https://doi.org/10.48550/arXiv.2305.17118.

Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B., Song, Z., Shrivastava, A., Zhang, C., Tian, Y., Ré, C., and Chen, B. Deja vu: Contextual sparsity for efficient llms at inference time. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 22137–22176. PMLR, 2023b. URL https://proceedings.mlr.press/v202/liu23am.html.

Mangrulkar, S., Gugger, S., Debut, L., Belkada, Y., Paul, S., and Bossan, B. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Wong, R. Y. Y., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. Specinfer: Accelerating generative LLM serving with speculative inference and token tree verification. *CoRR*, abs/2305.09781, 2023. doi: 10.48550/ARXIV.2305.09781. URL https://doi.org/10.48550/arXiv.2305.09781.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

ShareGPT. ShareGPT: Share your wildest ChatGPT conversations with one click., 2023. URL https://sharegpt.com/.

Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 322–337, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359658. URL https://doi.org/10.1145/3341301.3359658.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single GPU. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 2023. URL https://proceedings.mlr.press/v202/sheng23a.html.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

Thakkar, V., Ramani, P., Cecka, C., Shivam, A., Lu, H., Yan, E., Kosaian, J., Hoemmen, M., Wu, H., Kerr, A., Nicely, M., Merrill, D., Blasig, D., Qiao, F., Majcher, P., Springer, P., Hohnerbach, M., Wang, J., and Gupta, M. Cutlass, 1 2023. URL https://github.com/NVIDIA/cutlass/tree/v3.0.0.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023.

Williams, S., Waterman, A., and Patterson, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, apr 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings*

*of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 38087–38099. PMLR, 2023a. URL https://proceedings.mlr.press/v202/xiao23c.html.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. *CoRR*, abs/2309.17453, 2023b. doi: 10.48550/ARXIV.2309.17453. URL https://doi.org/10.48550/arXiv.2309.17453.

Ye, Z. FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer, 2023.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Zaken, E. B., Goldberg, Y., and Ravfogel, S. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In Muresan, S., Nakov, P., and Villavicencio, A. (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 1–9. Association for Computational Linguistics, 2022. doi: 10.18653/V1/2022.ACL-SHORT.1. URL https://doi.org/10.18653/v1/2022.acl-short.1.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C. W., Wang, Z., and Chen, B. $H_2o$: Heavy-hitter oracle for efficient generative inference of large language models. *CoRR*, abs/2306.14048, 2023. doi: 10.48550/ARXIV.2306.14048. URL https://doi.org/10.48550/arXiv.2306.14048.

Zhao, M., Lin, T., Mi, F., Jaggi, M., and Schütze, H. Masking as an efficient alternative to finetuning for pretrained language models. In Webber, B., Cohn, T., He, Y., and Liu, Y. (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pp. 2226–2241. Association for Computational Linguistics, 2020. doi: 10.18653/V1/2020.EMNLP-MAIN.174. URL https://doi.org/10.18653/v1/2020.emnlp-main.174.

Zhou, Z., Wei, X., Zhang, J., and Sun, G. PetS: A unified framework for Parameter-Efficient transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 489–504, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-11. URL https://www.usenix.org/conference/atc22/presentation/zhou-zhe.