DeltaZip: Efficient Serving of Multiple Full-Model-Tuned LLMs

Xiaozhe Yao
xiaozhe.yao@inf.ethz.ch
ETH Zurich
Switzerland

Qinghao Hu
qinghao@mit.edu
MIT
United States

Ana Klimovic
aklimovic@ethz.ch
ETH Zurich
Switzerland

Abstract

Fine-tuning large language models (LLMs) greatly improves model quality for downstream tasks. However, serving many fine-tuned LLMs concurrently is challenging due to the sporadic, bursty, and varying request patterns of different LLMs. To bridge this gap, we present DeltaZip, an LLM serving system that efficiently serves multiple full-parameter fine-tuned models concurrently by aggressively compressing model deltas by up to 10× while maintaining high model quality. The key insight behind this design is that fine-tuning results in small-magnitude changes to the pre-trained model. By co-designing the serving system with the compression algorithm, DeltaZip achieves 2× to 12× improvement in throughput compared to the state-of-the-art systems.

CCS Concepts: **• Computer systems organization → Cloud computing**; **• Computing methodologies → Machine learning**.

Keywords: Machine Learning Systems, Model Compression, Quantization, Sparsity, Hardware Acceleration, Model Inference and Serving

ACM Reference Format:
Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. 2025. DeltaZip: Efficient Serving of Multiple Full-Model-Tuned LLMs. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3689031.3717468

1 Introduction

Large Language Models (LLMs), such as GPT [58], Llama [74] and Gemini [72] have demonstrated remarkable performance and are widely used in a variety of applications, such as chatbots [56] and coding assistants [17, 35, 96]. To achieve high accuracy for a target domain, LLMs are first pre-trained on a large corpus of text data, then fine-tuned on application-specific tasks or datasets [50, 65], such as code [17], conversations [60], and human preferences [64]. Cloud AI infrastructure companies, such as OpenAI [55], Google [62], Microsoft [51], and Anyscale [8] expose APIs for users to fine-tune a pre-trained LLM with their own data and deploy the resulting customized model variant for inference.

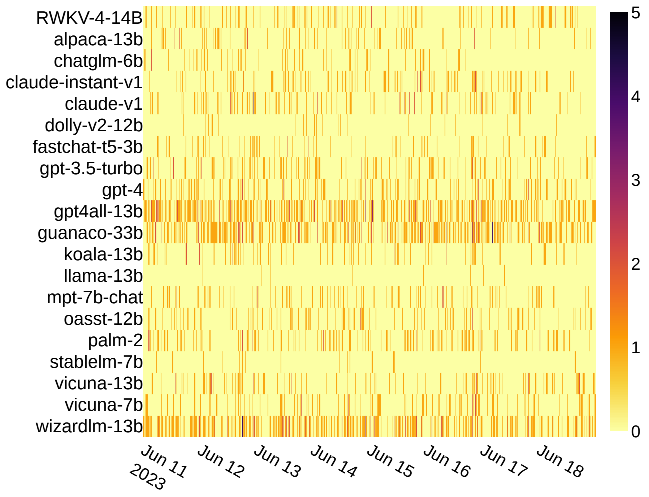


Figure 1. Invocation counts per 5-min time windows for 20 different models in the LMSys Chatbot Arena [92] trace.

While fine-tuning is typically a one-time effort performed off the critical path, LLM serving is critical to optimize as it is typically recurring and latency-critical. Techniques such as continuous batching [87], paged attention [43], prompt processing disaggregation [61, 70, 94], and tensor parallelism [46] optimize inference latency and throughput for individual models. However, the growing need to serve many model variants concurrently presents additional challenges. The need is particularly pronounced for LLM service providers, such as DeepInfra [22], Together.AI [73], or Fireworks.AI [27], who serve not only the popular base model, but also various fine-tuned models from either customers or the community.

On these service providers, the popularity of each model varies significantly between base models and their fine-tuned variants. From OpenRouter statistics [59], while the base model (*Llama-3.1-70B*) consumes ~1 billion tokens per day, the deployed fine-tuned variants consume much less, e.g., ~100 million for *Nemotron-70B* and *Hermes-3-70B*, and less than 10 million for *Sonar-70B* and *Lumimaid-70B*. In addition,

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3717468

Figure 1 shows that requests to base models and their variants can range from sporadic to very dense. For example, the models *alpaca-13b* [4], *koala-13b* [34], *gpt4all-13b* [6], *vicuna-13b* [93] and *wizardlm-13b* [83] are all fine-tuned variants of the base model *llama-13b*. We observe how certain variants, such as *wizardlm-13b* and *gpt4all-13b*, are highly popular over time while others, such as *alpaca-13b* and *vicuna-13b*, have sporadic requests. The varying popularity may be due to different accuracy on certain downstream tasks, and the irregular request patterns make it difficult to batch requests and decide how to provision GPU resources for each model.

Dedicating GPUs for each model variant minimizes latency, but requires many expensive GPUs that would sit idle for long periods of time (yellow areas in Figure 1). On the other hand, swapping model variants on a limited pool of GPUs reduces cost and improves utilization, but adds latency on the critical path of requests. Optimizing the storage hierarchy to reduce swapping latency can help [32], but requests still experience long queuing delays due to limited batching opportunities when treating each model variant as a separate model.

For LLM service providers, serving this long tail of less popular models cost-efficiently while maintaining low latency and high quality is a key challenge. The state-of-the-art approach to LLM serving with the proliferation of fine-tuned model variants is to adopt an entirely new fine-tuning paradigm, *parameter-efficient tuning* (PEFT). Instead of tuning model weights directly, PEFT tunes a compact, adjunctive model adapter (i.e., a small set of extra parameters) that is attached to the model for serving. For example, low-rank adaptation (LoRA) [39] is a popular PEFT method that freezes models weights and attaches low-rank matrices to the model structure which are fine-tuned on task-specific data. Systems like Punica [16] and S-LoRA [68] leverage the small size of LoRA adapters and the common base model weights to efficiently swap adapters and batch requests to optimize inference latency and throughput. Given the needs for deploying fine-tuned LLMs, these solutions have been integrated into commercial platforms such as LoRAX [63].

However, PEFT serving systems are not compatible with the traditional fine-tuning approach, *full model tuning* (FMT). While PEFT methods have achieved high accuracy for downstream tasks like SQL generation [8, 10] and ViGGO [42], they are still not able to match the accuracy of FMT for more complex tasks, such as coding and math [12], or when the fine-tuning dataset is particularly large [89]. Figure 2 summarizes the results of two recent studies [8, 12] comparing LoRA and FMT accuracy on three downstream tasks. Full-parameter fine-tuning remains appealing to applications that aim to maximize accuracy on more complex tasks and is still widely-used. Yet the serving solutions available for this paradigm (which either dedicate GPUs for each model variant or swap entire models) are either expensive or slow.

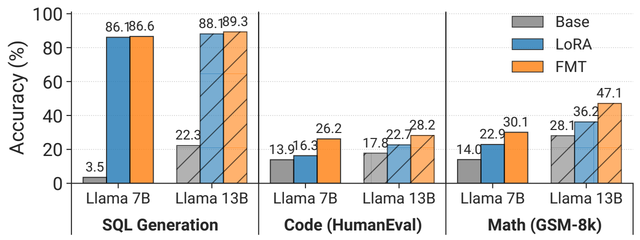


Figure 2. LoRA vs. full-model fine-tuning accuracy [8, 12]. LoRA fine-tuning is comparable for some tasks (SQL), but has lower quality on more complex tasks (Math and Code).

We aim to extend PEFT-based serving systems to also support efficient FMT model variant serving. Our key insight is that FMT model weights often have low-magnitude perturbations with respect to the original pre-trained model (see Figure 3), allowing us to aggressively sparsify, quantize, and compress *model deltas* while maintaining high accuracy. Due to their compact size, low-precision and sparse deltas can be swapped and served with low latency. We apply this idea in the design of DeltaZip, a multi-variant LLM serving system. DeltaZip extracts and compresses model deltas with ΔCompress, an algorithm we propose to help maintain high accuracy during compression. To efficiently serve FMT model variants, DeltaZip decouples base model serving and low-precision delta serving, inspired by how S-LoRA [68] and Punica [16] serve LoRA adapters. This decoupling enables DeltaZip to batch requests to different model variants that share the same base and perform low-precision inference for model deltas to minimize latency and memory bandwidth pressure on the GPU. We further optimize delta inference by designing a custom GPU kernel, Selective Batched Matrix Multiplication (*SBMM*), which selectively batches requests to the same delta to minimize random accesses and performs operations for multiple deltas in parallel to amortize kernel launch overhead. We build DeltaZip on top of vLLM [43] (which supports S-LoRA/Punica-based LoRA-serving [15, 68]) and adapt continuous batching [87] and model parallelism [69] for delta-based model serving.

To the best of our knowledge, DeltaZip is the first serving system to support both FMT and PEFT model variants while accelerating FMT model serving with hardware-optimized delta compression. In summary, our key contributions are:

- We propose ΔCompress (§4), a *hardware-efficient* compression algorithm that aggressively compresses model deltas post full-model fine-tuning. It applies structured sparsity, quantization, and optionally lossless compression. ΔCompress can compress a 70B-parameter Llama-2 model delta by 13× while maintaining comparable accuracy to the original FMT model. In contrast, applying a similar sparsification and quantization technique like SparseGPT [29] directly on the fine-tuned model substantially degrades accuracy, even with only 6× compression.

- We design DeltaZip (§5), a serving system that leverages ΔCompress to efficiently serve FMT model variants. It decouples and parallelizes base and delta model inference, reducing queuing delays by batching requests to different model variants derived from the same base and leveraging our custom GPU kernel for hardware-optimized low-precision and sparse delta serving. DeltaZip achieves 2× to 12× higher throughput compared to vLLM [43].
- We identify challenges when deploying DeltaZip in practice and propose solutions to address them. In particular, we study how many deltas to place concurrently to balance GPU memory usage and how to reduce starvation when serving many model deltas.

Our compression pipeline and serving system is open source and available at https://github.com/eth-easl/deltazip.

2 Background and Motivation

We first introduce key concepts in LLM fine-tuning and serving, then highlight key challenges and opportunities.

2.1 Background

LLM fine-tuning. In contrast to prior deep learning workloads that follow a *task-specific* paradigm, which trains a model from scratch on domain-specific data to tackle a particular task (e.g., machine translation [71, 81]), LLMs follow a *pretrain-then-finetune* paradigm, which pretrains a model on a massive general-domain corpus and then fine-tunes it for specific objectives. For example, ChatGPT is fine-tuned to follow human instructions [60]. There are two main fine-tuning approaches: 1) *Full Model Tuning* (FMT) which updates all model parameters and 2) *Parameter-Efficient Tuning* (PEFT) which adds a small number of extra parameters after pretraining, called *adapters*, e.g., low-rank matrices learned during fine-tuning. PEFT methods, such as LoRA [39], are popular ways to reduce the compute and memory requirements of both fine-tuning and serving. However, the choice of fine-tuning paradigm impacts accuracy. While PEFT methods can achieve high accuracy for a variety of tasks [39], recent studies [8, 12, 89] — summarized in Figure 2 — reveal that FMT still achieves higher accuracy for more complex tasks.

LLM compression. Model compression is a popular approach to reduce the memory and compute requirements of LLM inference in resource-constrained environments. Techniques like GPTQ [31], SparseGPT [29] and AWQ [47] reduce memory footprints and improve latency while maintaining model quality (when applied in moderation). Pushing these techniques to extremely low bit-width quantization and sparsity, such as 2-bit quantization and more than 50% sparsity, results in significant model quality degradation [14, 49].

LLM serving. The LLM inference involves two phases: (1) *prompt processing* (prefill), where the tokens (i.e., basic units of text) in the input prompt are processed in parallel. This phase can be parallelized since all previous tokens are known from the user-provided prompt, and it is usually compute-bound. (2) *token generation* (decode), where the model iteratively generates one token for each forward pass. Due to the inter-token data dependency, this phase cannot be parallelized and is typically memory-bound. Token generation stops when the model generates an end-of-sequence (EOS) token or meets a user-defined condition (e.g., reach the maximum number of generated tokens). Many works [16, 43, 87], including our own, focus on optimizing the token generation phase as it is the main bottleneck for LLM serving.

2.2 Challenges for Multi-Variant Model Serving

With the proliferation of fine-tuned LLM variants, each specialized for a particular user's task or domain, it is critical to design LLM serving systems that can efficiently multiplex requests to different model variants. Yet, on today's commercial LLM serving platforms, fine-tuned model variant serving is still more expensive than base model serving [1, 57]. The high cost is due to several key challenges that lead to low resource utilization:

Low request rates per model variant. A naive approach to serve multiple fine-tuned models is to consider each model as a separate model and dedicate a group of GPUs for each variant. However, as shown in Figure 1, the requests to each model variant are sporadic and often low in volume, which limits the batch size. Hence, dedicating GPUs for each model underutilizes resources and leads to high cost.

Swapping incurs high latency. Another approach is to swap the models in and out of a limited pool of GPUs. However, the unpredictable nature of model invocations makes it hard to predict when a request for a particular model will arrive to enable prefetching. Swapping models in and out of GPUs on the critical path of requests leads to high latency and GPUs also remain underutilized.

Accuracy gap between PEFT and FMT. Although PEFT methods such as LoRA [39] produce significantly smaller fine-tuned adapters and systems [15, 68, 80, 97] have been built for serving adapters, these methods face the challenge of model quality. As shown by recent studies [8, 12, 89] (summarized in Figure 2), FMT still achieves higher accuracy for more complex tasks compared to PEFT methods.

3 DeltaZip Overview

We observe three key opportunities to address the aforementioned challenges and improve the efficiency of FMT serving (§3.1). We then describe how we leverage these opportunities to design DeltaZip, a multi-variant LLM serving system that complements parameter-efficient model serving with hardware-efficient full model tuning serving (§3.2).

3.1 Opportunities and Key Insights

1. Model deltas are highly compressible. Although fine-tuning can significantly improve model performance on specific tasks, we find it typically results in small-magnitude

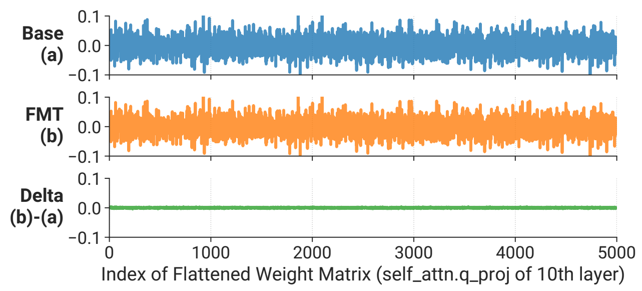


Figure 3. Flattened weight matrix in an intermediate layer of the pre-trained model (a), the fine-tuned model (b), and the model delta between the two (bottom, (b)-(a)).

changes to model parameters. Figure 3 shows the distribution of the weight matrix $\mathbf{w_q}$ in a transformer layer of a pre-trained `Llama-2-7b` model, its fine-tuned counterpart `Vicuna-7b-v1.5` model, and the model delta, which is obtained by subtracting the base model from the fine-tuned model. The delta has significantly smaller magnitude range and fewer outliers. This makes the delta easier to compress than the model itself, with both quantization [31, 47] and sparsification [29]. As quantization involves calculating the maximum value such that the quantization preserves the outliers in the weight matrix, a more concentrated distribution results in a denser quantization grid, which can maintain model quality while using lower bit width. In addition, the prevalence of near-zero values makes it easier to apply sparsification to the model delta than to the full model. We will show that this enables us to compress model deltas by over 10×, making it fast to swap and serve while achieving comparable quality to full-precision model serving.

2. Many model variants share the same base model, for which we can batch many requests. Since pre-training LLMs requires immense datasets and compute resources, only a handful of organizations are well-positioned to produce high-quality pre-trained models [3, 72]. Instead, most organizations rely on fine-tuning pre-trained models for their use cases, as this is typically much cheaper [1]. This means that fine-tuned models often share the same base model, even if they are used for different applications. For example, GitHub's Copilot [24] and OpenAI's ChatGPT [60] are both fine-tuned from GPT models. Hence, although batch sizes for individual model variants may be limited (Figure 1), we can quickly accumulate large batches to common base models to improve GPU utilization and reduce queuing delays. This requires decoupling base model and delta serving, which we will discuss in §5.1.

3. GPU support for delta computations. We can accelerate delta inference by leveraging features of modern GPU hardware, such as: 1) using sparse tensor cores [11] for sparse delta computations, 2) performing multiple low-precision matrix multiplications in parallel to improve stream multi-processors (SMs) utilization, and 3) reducing memory bandwidth pressure for delta matrix computations by designing a custom GPU kernel that minimizes data movement from GPU global memory to device memory.

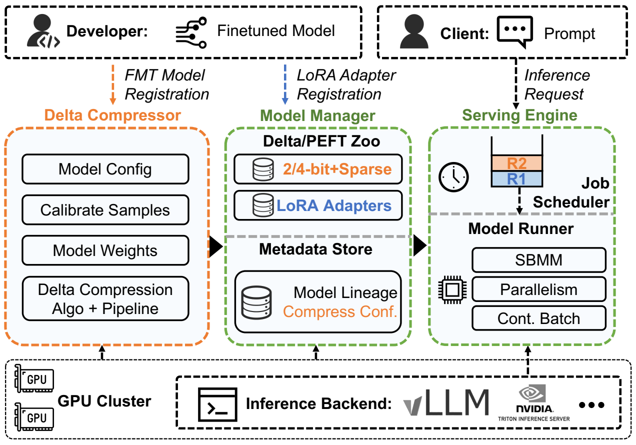


Figure 4. DeltaZip system architecture.

3.2 System Overview

To realize the above opportunities in practice, we propose DeltaZip. Our key contributions are a model delta compression algorithm (Δ*Compress*) and the end-to-end DeltaZip serving system. The algorithm compresses FMT deltas into a hardware-efficient, low-precision, and sparse format that preserves high accuracy, while the serving system incorporates state-of-the-art serving optimizations and adapts them for low-latency, high-throughput inference with model deltas. DeltaZip is designed for LLM service providers to operate, and LLM users can interact with the system through the standard HTTP API frontend, as if using a dedicated LLM.

System architecture. Figure 4 shows the DeltaZip system architecture. The system consists of three main components: 1) the *Delta Compressor* (§4) which extracts and compresses the delta from a FMT model registered by the user, 2) the *Model Manager* which tracks and manages model metadata, and 3) the *Serving Engine* (§5) which serves inference requests for base and fine-tuned models.

Life of a request. The model developer first uploads a fine-tuned model to the Delta Compressor, together with some metadata (such as the pre-trained model identifier) and a small calibration dataset that the compression algorithm uses to measure and minimize accuracy loss. The compressor computes and compresses the model delta, then stores it in a packed, low-precision, and sparse format in the Model Manager's delta zoo. The manager keeps track of metadata for each stored delta, such as its compression configuration (such as the bit width per parameter and sparsity level) and model lineage. In addition to deltas, the model manager also allows developers to register LoRA adapters directly, and the serving engine can serve them as well.

The Serving Engine serves inference requests for fine-tuned models whose deltas are stored in the model manager. Users send inference requests to the engine's API frontend,

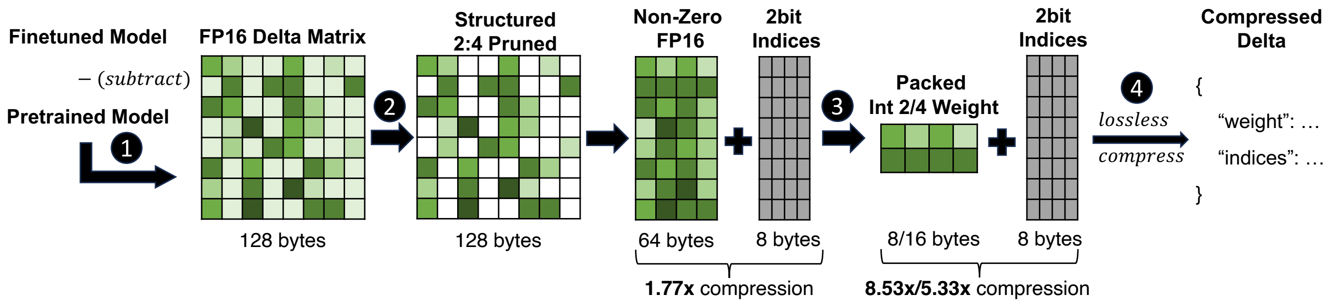


Figure 5. DeltaZip Compression Pipeline. The pipeline consists of delta extraction, sparsification & quantization, and optionally lossless compression. The compressed delta is stored as a dictionary of compressed weight matrices and metadata.

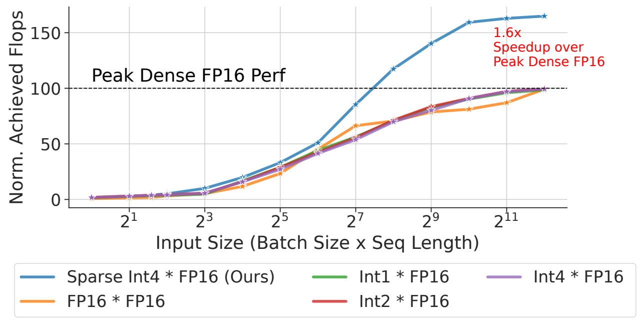


Figure 6. (Compressed) Matrix Multiplication Performance

which fetches the requested deltas into CPU main memory, if they are not already present in CPU or GPU memory. Meanwhile, the frontend forwards the request to the job scheduler, which queues the requests. The job scheduler performs continuous batching by assigning requests to the model runner per-iteration (i.e., for each forward pass of the model). The model runner is responsible for processing the batched requests. Internally, the model runner can process requests for different FMT and PEFT models in parallel, by decoupling base model and delta (or PEFT adapter) inference computations. The model runner also leverages tensor parallelism and supports large models that do not fit in a single GPU.

4 Model Delta Compression

When model developers register an FMT model into the system, the Delta Compressor computes the delta and applies a compression pipeline (§4.1) to reduce the size of the model state. The compression pipeline only runs when a developer registers a new model. We develop the ΔCompress algorithm (§4.2) to apply quantization and sparsification in the compression pipeline in a way that preserves high accuracy.

We design DeltaZip to be agnostic to the compression pipeline, such that users can apply a variety of compression techniques to model deltas, such as GPTQ [31], SparseGPT [29], and AWQ [47]. Here we describe a SparseGPT-inspired compression pipeline, which we use in our implementation and evaluation of DeltaZip. We implemented this compression pipeline on top of AutoGPTQ [78] and SparseGPT [29].

4.1 DeltaZip Compression Pipeline

Within DeltaZip, the Delta Compressor encompasses multiple steps as shown in Figure 5. Initially, Step 1 computes the model delta by subtracting the pre-trained base model weights from the fine-tuned model. The remaining steps are applied on the delta matrix since it has a smaller range of values (see Figure 3) and is hence more amenable to compression compared to the fine-tuned model weight matrix.

Step 2 applies structured 2:4 pruning [40, 95], which involves setting at least two elements among each group of four contiguous elements to 0 in the delta matrix. With structured pruning, DeltaZip only stores the non-zero values of the delta matrix and their 2-bit indices. The main motivation for applying structured sparsity is due to its *hardware-efficiency*. Compared to quantization only or unstructured sparsity, the particular pattern we used in structured sparsity enables higher peak performance with large input size, since modern NVIDIA GPUs from the Ampere architecture onwards and AMD RDNA/CDNA GPUs have specialized hardware support for this feature [5, 11]. This hardware support is automatically enabled when running compatible kernels. Given the trend toward hardware optimization, we expect future GPUs will continue to support this feature.

Figure 6 shows a microbenchmark of matrix multiplication performance with different compression configurations. We observe that with small input sizes (e.g., from 1 to 4, which is common during the decode phase), structured sparse matrix multiplication achieves similar performance to quantization-only compression and outperforms the uncompressed version. This is mainly because both sparsity and quantization reduce data movement between GPU HBM and compute unit. However, with large input sizes (e.g., from 16 to 4096, which is typical during the prefill phase), structured sparsity significantly outperforms quantization-only compression, as structured sparsity leverages the sparse tensor cores [11] on GPUs to achieve higher performance.

Step 3 quantizes the pruned delta weight matrix, which squeezes the values into a smaller bit-width format and packs the data. For example, 4-bit quantization packs 8 values into a single 32-bit value, achieving 4× compression ratio.

Algorithm 1 ΔCompress algorithm. Given an N-layer FMT model where each layer has a weight matrix $\mathbf{w}_f$ of the shape $d_{\text{row}} \times d_{\text{col}}$, and the corresponding layer in the base model has a weight $\mathbf{w}_b$, the algorithm computes quantized weight $\mathbf{Q}$ and pruning mask $\mathbf{M}$. $\odot$ denotes elementwise multiplication.

```
1: for $n = 0, 1, 2, \cdots, N$ do
2:     $\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$          ▷ Binary Pruning Mask
3:     $\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$          ▷ Quantized Delta
4:     $\Delta = \mathbf{w}_f - \mathbf{w}_b$                          ▷ Extract Delta
5:     $\mathbf{Q}, \mathbf{M} = \text{Compress}(\Delta, \mathbf{X}_n)$          ▷ e.g., SparseGPT
6:     $\tilde{\mathbf{w}}_q \leftarrow \mathbf{Q} \odot \mathbf{M} + \mathbf{w}_b$          ▷ Reconstruct Weight
7:     $\mathbf{X}_{n+1} = \tilde{\mathbf{w}}_f \mathbf{X}_n$ as input of next layer.
8: for $n = 0, 1, 2 \cdots, N$ do
9:     pack and store $\mathbf{Q}, \mathbf{M}$ of layer $n$.
```

Step 4 is an optional final step that applies lossless compression. We use the GDeflate algorithm from `nvcomp` [54] for lossless compression for fast decompression on GPUs. This step is beneficial when the disk bandwidth is a bottleneck (such as with NFS). In such cases, users can opt-in to lossless compression to reduce the disk I/O time. If disk I/O is not a bottleneck, lossless compression may not be beneficial due to the decompression overhead.

Hardware-efficient design. We design DeltaZip's compression pipeline with GPU hardware features in mind. Structured pruning (Step 2) leverages sparse tensor cores [11] for fast sparse matrix multiplication. The quantization (Step 3) allows us to move a smaller amount of data between GPU global memory to device memory, alleviating the memory bandwidth bottleneck. Lossless compression (Step 4) uses GPU decompression engines [54].

4.2 ΔCompress Algorithm

The core of the compression pipeline is the lossy compression algorithm that finds the optimal pruning mask (in Step 2) and quantized weight matrix (in Step 3). We design ΔCompress to compress the model deltas in a way that minimizes the loss between the outputs computed by the original weights and the compressed weights. ΔCompress achieves this by calibrating the compression algorithm with a subset of the training dataset, provided by model developers when registering the FMT model to DeltaZip. As outlined in Algorithm 1, ΔCompress iteratively (Line 1) compresses the delta (extracted in Line 4) for each layer of the model. For each layer, the objective is to find the optimal pruning mask $\mathbf{M}$ and quantized weight matrix $\mathbf{Q}$ (Line 2, Line 3). ΔCompress is designed to be compatible with various different compression techniques to achieve this goal, such as GPTQ [31], SparseGPT [29] and AWQ [47]. In our current implementation, we follow the optimal brain surgeon [38, 44] approach and leverage SparseGPT [29] (since it has support for both sparsification and quantization) to compute the optimal $\mathbf{M}$ and $\mathbf{Q}$ by solving the following optimization problem:

$$\arg\min_{\tilde{\Delta}} ||\Delta \cdot \mathbf{X} - \tilde{\Delta} \cdot \mathbf{X}||_2^2 \quad (1)$$

where $\tilde{\Delta}$ is the compressed delta and $\mathbf{X}$ is the input to the layer, which is from the calibration set used for compression.

The major distinction from full model compression is that ΔCompress reconstructs the weight matrix for each layer (Line 6) after compressing the delta and computes the input for next layer (Line 7). This is because the input data (i.e., $\mathbf{X_n}$ in Line 5) is crucial for the compression algorithm. Without re-adding the base weight matrix and reconstructing the weight matrix for each layer, the magnitude of the deltas causes diminishing outputs, leading to vanishing activations in deeper layers. The vanishing activations, as the input to the next layer, will make the compression algorithm fail to capture the input (i.e., X_n Line 5). By extracting the delta and reconstructing the weight for each layer on the fly, ΔCompress ensures proper calibration and maintains high model quality.

Beyond model quality, ΔCompress has three advantages: 1) **Low memory requirement**. As ΔCompress performs layer-wise compression, it only needs a GPU with sufficient memory for a single layer to perform the forward pass.
2) **No need to retrain the model**. Unlike some other compression algorithms [14, 75], ΔCompress does not require further fine-tuning the model to recover the model quality.
3) **Data-Efficiency.** ΔCompress requires a small calibration set to achieve high accuracy, which does not need to be the exact, entire training set. In our experiments, we found 256 samples chosen from open-source instruction tuning dataset `UltraChat` [23] are sufficient for calibration. In practice, if the users fine-tune their models through the service provider, then the training data can be directly used as the calibration set. If the users fine-tune their models on their own, they can use a small, representative subset (e.g., by uniform sampling) of the training data as the calibration set. The users can also evaluate the model quality after compression using an evaluation set to ensure the quality meets their requirements, and adjust the calibration set if needed.

Depending on the model size and hardware, the compression process takes some time to run. From our experiments, compressing a 7B model takes around 30 minutes on a single RTX 3090 GPU and AMD EPYC 7313 16-Core CPU. Although this still introduces extra cost and overhead, the process is offline (i.e., not on the critical inference path) and only runs once when the model is registered, making it acceptable.

5 Serving System Design

We implement DeltaZip in 18K lines of Python and 1K lines of C++/CUDA code. We build the serving engine on top of vLLM [43], HuggingFace Transformers [79], and Sparse Marlin [13]. §5.1 explains how DeltaZip decouples base and delta serving to maximize request batching for base model inference. §5.2 describes how the system parallelizes low-precision delta serving with a custom GPU kernel design. Finally, we describe how we extend model parallelism for delta serving (§5.3) and schedule delta serving requests with continuous batching (§5.4).

5.1 Base and Delta Decoupling

DeltaZip's serving engine always keeps the base model in GPU memory and swaps compressed deltas on-demand to serve inference requests. The naive approach to serve a fine-tuned model is to load its compressed delta, decompress it, add it to the base model, and then perform inference. However, this approach is inefficient for several reasons: 1) it requires decompressing the delta on the critical path, which adds latency; 2) it does not allow batching requests to different model variants with the same base model; 3) performing inference after adding the delta back to the full-precision base model does not leverage low-precision computation to reduce delta inference latency; and 4) storing decompressed deltas in GPU memory to add back to the base model limits how many deltas can fit concurrently in the GPU.

To improve inference latency and throughput, our first optimization is to **decouple the inference computation of the base model and delta**. Consider a matrix multiplication, which we can decouple as follows by the distributive law:

$$\begin{aligned} \mathbf{Y} &= \mathbf{w}_{\text{fine-tuned}}\mathbf{X} = (\mathbf{w}_{\text{base}} + \Delta)\mathbf{X} \\ &\approx \underbrace{\mathbf{w}_{\text{base}} \cdot \mathbf{X}}_{\text{Batched FP16 matmul}} + \underbrace{\Delta \cdot \mathbf{X}}_{\text{Quantized and sparse matmul}} \end{aligned} \tag{2}$$

In Eq 2, $\mathbf{w}_{\text{base}}\mathbf{X}$ refers to the matmul with the base model, which is shared across all fine-tuned models and we can compute this with a standard `GEMM`. $\Delta\mathbf{X}$ denotes the computation with the delta, which is in a sparse and low-precision format. DeltaZip decouples the computation into two parts and executes the batched base model and low-precision delta matrix multiplications independently and in parallel.

Since the distributive law does not hold for non-linear operations, such as activation functions, we decouple the computation at the granularity of linear layers. We merge results from the base model and the delta part after each linear layer to get the output to feed into a non-linear operation. In a transformer block, we serve all linear layers with low-precision, such as the QKV projections $\mathbf{w_q}, \mathbf{w_k}, \mathbf{w_v}$, output projection $\mathbf{w_o}$ and the linear layers in the MLP module.

Decoupling base and delta serving improves GPU utilization and performance in several ways. First, for the base model computation, it enables batching requests for different model variants, as long as they share the same base model. Second, for the delta computation, keeping deltas in low-precision, sparse formats allows us to fit more deltas in GPU memory and reduce swapping. Third, low-precision delta serving also reduces inference latency since token generation in LLM inference is inherently memory-bound (as

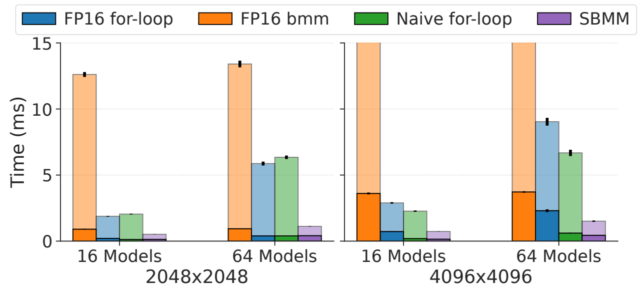


Figure 7. Breakdown of total execution time for different implementations of batched matrix multiplication. The dark part of the bar shows the portion of total execution time spent on computation. Naive for-loop refers to performing low-precision and sparse computation while looping through all models. SBMM refers to our proposed kernel.

discussed in §2.1), so the decoding latency is proportional to the GPU memory consumption of the model weights.

If there are M base models and $M > 1$, we divide the GPU cluster into M sets of GPUs, each dedicated to serving a particular base model and its fine-tuned variants. Hence, even with many model variants, we only need M sets of GPUs to serve all these models. LoRA serving systems [15, 68] have a similar assumption. However, operators can also deploy multiple base models per GPU if there is sufficient memory.

5.2 GPU-Efficient Delta Serving

DeltaZip parallelizes the delta computation (ΔX) in a GPU-efficient manner with an approach we call Selective Batched Matrix Multiplication (*SBMM*). For a single batch of requests $\mathbf{X}_i^j$ where i is the request index and j is the delta index, the delta computation can be formalized as: given a batch of requests $\mathbf{X}_1^1, \mathbf{X}_2^2, \ldots, \mathbf{X}_i^j$, compute $\mathbf{Y}_1 = \mathbf{X}_1^1 \cdot \Delta_1, \ldots, \mathbf{Y}_i = \mathbf{X}_i^j \cdot \Delta_j$. We use an index Idx_i to denote the delta index for each request i. Naively, we can loop through different deltas in the batch, find the respective requests and compute the matrix multiplication. However, we observe that this approach is inefficient for two main reasons: 1) it introduces random memory accesses to fetch the inputs and write the outputs to the correct locations, 2) it needs to launch the matrix multiplication kernel multiple times which computes on a small number of requests each time, incurring a high overhead and low GPU utilization. Another option is to use an operator with a batch dimension like `torch.bmm` [28]. However, this requires first stacking the weight matrices for each input into a single matrix, which is not efficient or scalable with delta matrices, as they have large memory footprints. Figure 7 shows the total execution time of different batched matrix multiplication implementations and the portion of time each spends on computation. We find that although the low-precision matmul kernel reduces computation time, the total execution time is still high as it is dominated by kernel launch time and other overheads.

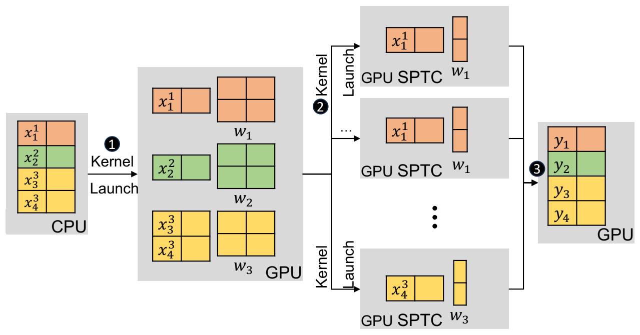


Figure 8. SBMM kernel launch. The kernel computes the output $y_i = \text{matmul}(x_i, w_{Idx_i})$ where the matmul is low precision and sparse. Different colors represent different deltas and their respective requests. SPTC=sparse tensor cores.

To reduce this overhead and improve GPU utilization, we apply two optimizations. First, before launching a computation on the GPU, DeltaZip's job scheduler **reorders the requests to group requests belonging to the same delta together**. This reduces random data accesses during computation and enables higher batch sizes for delta serving.

Second, we **design a kernel that performs the SBMM operation for multiple deltas in a single kernel launch**. We design SBMM to compute its kernel launch configuration dynamically to balance resources for each delta, since there is often a different number of requests for each delta in the batch. We implement this with CUDA dynamic parallelism [7] on modern GPUs. Specifically, SBMM first launches a kernel that prepares the launch configuration, pointers to the weight, input and output addresses, and other necessary information for each delta, and then launches the actual blocked matmul kernel, which fuses dequantization for each delta and leverages sparse tensor cores on GPUs. The dynamic parallelism feature is revamped in the recent CUDA toolkit [9] and offers substantial performance improvements. Figure 8 illustrates the kernel launch process for 3 deltas and 4 requests where the third delta has two requests. The first kernel is launched from the host and prepares the addresses of the weights, input and output for each delta, and then launches a blocked matmul for each delta in the second step. The actual matmul, in the last step, writes the results to the output. Figure 7 shows that even though the actual computation time is similar, our optimized kernel significantly reduces overhead and improves end-to-end latency.

We design our SBMM kernel to be compatible with popular low-precision and sparse matrix multiplication implementations. Optimizing such computations is an active research area for which many libraries have been developed, such as BitBLAS [77], Marlin [30], and SparseMarlin [13]. Maintaining compatibility with these libraries allows us to leverage the latest hardware and community efforts.

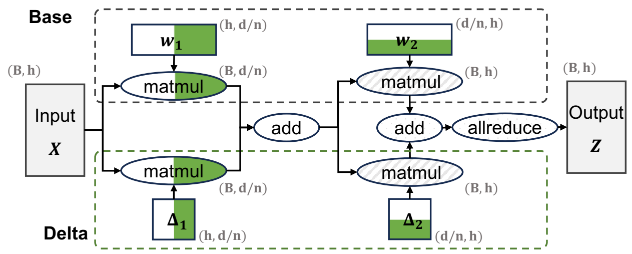


Figure 9. Tensor Parallelism in DeltaZip for $n = 2$ GPUs. B = number of tokens, h = input dimension, d = hidden size. Column-wise and row-wise partitions are illustrated as vertically and horizontally divided boxes, respectively.

5.3 Model Parallelism for Delta Serving

To serve large models that do not fit into a single GPU, DeltaZip extends Megatron-style [69] tensor parallelism to serve compressed deltas. In Megatron-style tensor parallelism, the model is partitioned column-wise or row-wise across multiple GPUs. DeltaZip adapts this to delta serving by **partitioning the delta in the same way as the base model**. We first illustrate how our partition strategy works in Figure 9 with two linear layers, and then explain how we extend this approach to self-attention layers. Figure 9 assumes that we have two GPUs, a base model with two linear layers $[\mathbf{w}_1, \mathbf{w}_2]$ and a delta $[\Delta_1, \Delta_2]$. In the upper box, we partition the base model $\mathbf{w}_1$ to $[\mathbf{w}_{1,1}, \mathbf{w}_{1,2}]$ by column across two GPUs, and calculate the partial results $Y_{\text{base},i} = X\mathbf{w}_{1,i}$ on each GPU. In the lower box, we partition the delta Δ_1 to $[\Delta_{1,1}, \Delta_{1,2}]$ in the same way, and calculate the partial results $Y_{\text{delta},i} = X\Delta_{1,i}$ on each GPU. The result of the matrix multiplication can be computed on each GPU independently without any synchronization with other GPUs as $Y = [Y_1, Y_2] = [Y_{\text{base},1} + Y_{\text{delta},1}, Y_{\text{base},2} + Y_{\text{delta},2}]$.

We then partition the second linear layer $\mathbf{w}_2, \Delta_2$ with row-parallel across two GPUs as $[\mathbf{w}_{2,1}, \mathbf{w}_{2,2}]^T$, $[\Delta_{2,1}, \Delta_{2,2}]^T$. Then the output of this layer becomes $Z = [Y_1, Y_2] \cdot [\mathbf{w}_{2,1} + \Delta_{2,1}, \mathbf{w}_{2,2} + \Delta_{2,2}]^T$. To compute this, we first perform $Y_i \cdot \mathbf{w}_{2,i}$ and $Y_i \cdot \Delta_{2,i}$ on each GPU individually, and then sum the results across GPUs with an all-reduce operation as the output.

In the self-attention module, we partition the $\mathbf{q}, \mathbf{k}$ and $\mathbf{v}$ projections as column-wise linear layers and the output projection $\mathbf{o}$ as row-wise linear layers. We then apply the same strategy as we described above to compute the output of the transformer block.

5.4 Continuous Batching and Scheduling with Delta

Last but not least, DeltaZip optimizes inference by extending continuous batching [87], a standard technique to improve GPU utilization for LLM serving. DeltaZip implements a software scheduler per set of GPU workers that form a tensor parallel serving group. At every iteration, the scheduler picks K requests to serve concurrently on a first-come-first-served basis. These K requests belong to at most N deltas that can fit concurrently in GPU memory. After selecting the K requests and N deltas, the scheduler searches the request queue for all requests that belong to the selected N deltas. The intuition behind this design is to maximize batching by allowing requests to *skip the line* if and only if they are for one of the N selected deltas. The scheduler then delivers the batch of requests to the Model Runner, which starts to load the requested deltas, performs prompt processing if needed, and then batches request decoding for different models.

Tuning N, the number of concurrent deltas. DeltaZip tunes N empirically based on a short trace profiling phase. We explore the impact of varying N. Intuitively, processing more deltas concurrently (larger N) allows for more extensive request batching, enhancing throughput. However, collocating many deltas increases GPU memory pressure, potentially degrading performance if there are already many requests to batch for a particular delta. We conduct a microbenchmark on RTX 3090 GPU and synthetic trace to study the effectiveness of offline profiling. On the left side of Figure 10, we show how N affects the serving latency when executing a 25 second time interval of a trace with arrival rate 3 and zipf-4.0 model popularity distribution. In this case, DeltaZip would pick $N = 3$ as it achieves optimal performance during profiling. We then run a series of traces under different settings. The right side of Figure 10 shows that $N = 3$ remains optimal or near-optimal across different arrival rates and model popularity distributions. We generally find that offline profiling with a short trace is sufficient to determine the (near-)optimal number of concurrent deltas. Dynamic tuning can also be implemented. If offline profiling is not possible, operators can heuristically set N. If there are only few requests per delta, we can allow more deltas (and hence more requests in parallel) to improve the GPU utilization. If there are many requests per delta, we can reduce N to avoid memory pressure. Like other systems, operators can also adjust the maximum batch size and concurrent deltas as hyperparameters.

Avoiding starvation. Allowing requests to skip the line is good for batching, but it may cause starvation if requests for the currently selected deltas keep arriving and skipping the line before the system has a chance to swap and serve other deltas. When a request skips the line to be batched with a currently loaded delta, we refer to the original request for that delta at the head of the queue as the *parent request*. To alleviate starvation, DeltaZip uses a preemption strategy: **requests that skip the line are preempted when their parent request finishes**. Preempted requests get reinserted in their original place in the request queue (as if they did not skip the line) and can get scheduled in the next iteration. DeltaZip currently swaps the intermediate state of preempted requests to CPU memory and resumes computation when the request is re-scheduled. Future work involves exploring whether and when recomputing from scratch may be faster than swap-and-resume.

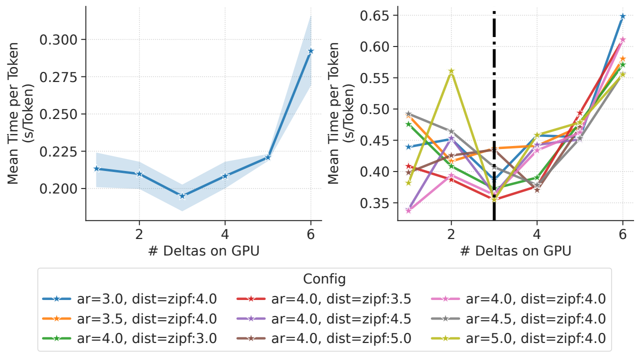


Figure 10. Normalized latency of DeltaZip with varying N, the number of concurrent deltas in GPU memory.

Scalability. DeltaZip employs a hierarchical management strategy when the number of deltas exceeds both GPU and host memory capacity. When there are more deltas than the capacity of GPU memory and even host memory, DeltaZip will swap deltas in and out of GPU/CPU memory to disk, and only load deltas when they are needed. For an inference batch, only a fixed and pre-set number of deltas (or less) will be processed concurrently and thus a single batch will not exceed the GPU memory capacity. The performance of DeltaZip will gracefully degrade if there are sporadic requests for deltas that reside on disk due to swapping.

6 Evaluation

6.1 Performance Evaluation Setup

Experiment testbed. We conduct our experiments on a homogeneous GPU cluster. Each node has 2× Intel Xeon 8358P CPUs (128 threads) and 2TB DDR4 memory. We use 4 A800 GPUs per node and GPUs are interconnected to each other by NVLink and NVSwitch [2]. Additionally, the cluster adopts an all-NVMe shared parallel file system, ensuring rapid data access and storage, connected through a 50Gbps RoCE network. All the experiments are conducted on this cluster unless explicitly stated.

Models and downstream tasks. We use the Llama-2 [74] model with 7B, 13B, and 70B parameter and their fine-tuned variants. For 7B and 13B models we use the `Vicuna-v1.5` [18] fine-tuned models since the fine-tuning data is disclosed and can be used for calibration. For 70B model we use the `Llama-2-70B-chat-hf` variant, and we use the fine-tuning dataset from `Vicuna` [18] as a proxy to calibrate the compression. We mainly focus on serving 7B and 13B models in the serving experiments and compress them to 4-bit with 50% sparsity. For these large models, we evaluate the post-compression quality on standard benchmarks in *lm-eval-harness* [33]. We also fine-tune a smaller scale `Pythia-2.8B` model on *natural instructions* tasks and evaluate the quality on the known downstream tasks. Since the accuracy drop

Base Model	Method	Downstream Tasks↑ T1	T2	T3	Compress Ratio
Pythia 2.8B	FP16	73.76	94.23	79.61	1.00×
	SparseGPT (4bit★)	67.59	91.99	72.67	3.93×
	DeltaZip (4bit★)	73.13	**94.30**	**79.52**	4.75×
	DeltaZip (2bit★)	**74.44**	94.22	78.90	**8.36×**
Llama 7B	FP16	80.92	41.65	27.34	1.00×
	SparseGPT (4bit★)	67.16	36.48	24.12	5.61×
	AWQ (4bit)	81.22	**42.24**	27.34	3.64×
	DeltaZip (4bit★)	81.41	41.72	27.50	5.39×
	DeltaZip (2bit★)	**81.56**	41.91	**28.26**	**10.36×**
Llama 13B	FP16	85.29	43.00	27.04	1.00×
	SparseGPT (4bit★)	79.88	35.01	23.20	5.91×
	AWQ (4bit)	84.80	**43.37**	**27.80**	3.82×
	DeltaZip (4bit★)	**85.11**	42.48	27.04	5.91×
	DeltaZip (2bit★)	84.95	42.54	27.65	**11.83×**
Llama 70B	FP16	86.73	44.25	33.18	1.00×
	SparseGPT (4bit★)	85.87	37.06	27.80	6.11×
	AWQ (4bit)	86.36	44.04	31.80	3.72×
	DeltaZip (4bit★)	**87.28**	**44.18**	32.87	5.84×
	DeltaZip (2bit★)	86.67	43.47	**33.49**	**13.96×**
Gemma 2 2B	FP16	83.70	45.10	27.65	1.00×
	SparseGPT (4bit★)	74.86	38.79	21.50	1.53×
	AWQ (4bit)	83.14	**44.97**	**28.11**	2.33×
	DeltaZip (4bit★)	**83.79**	44.24	27.50	3.26×
	DeltaZip (2bit★)	83.66	44.40	26.26	**4.08×**
Gemma 2 9B	FP16	88.87	51.75	33.18	1.00×
	SparseGPT (4bit★)	86.26	45.71	29.95	2.43×
	AWQ (4bit)	88.47	51.24	33.33	3.10×
	DeltaZip (4bit★)	**88.96**	**51.32**	**34.87**	4.61×
	DeltaZip (2bit★)	88.68	50.95	33.94	**7.50×**

Table 1. Model quality of DeltaZip vs. uncompressed (FP16) vs. SparseGPT [29] and AWQ [47]. For `Pythia`, T1, T2, T3 show the accuracy on 3 downstream tasks (Amazon Review Classification, Synthetic Palindrome Numbers, Yes/No Question) from *natural instructions* [52]. For other models, T1, T2, T3 show accuracy on 3 standard benchmarks (BoolQA, TruthfulQA, LogiQA) in *lm-eval-harness* [33]. ★ indicates 50% structured pruning in addition to quantization.

is already substantial for SparseGPT and we found that the accuracy drop is more significant with lower precision, we do not evaluate the 2-bit 50% sparsity for SparseGPT.

Workload traces. To evaluate the serving performance, we use the prompts and responses sampled from the LMSys Chatbot Arena [92]. We consider three types of model popularity distribution: 1) Uniform: all models are equally popular. 2) Skewed: model popularity follows a Zipf-α distribution. In other words, the popularity of the i-th model is proportional to $1/i^{\alpha}$. We set $\alpha = 1.5$ for the skewed distribution. 3) Azure trace: since there are no publicly available traces for multi-variant LLM serving available, following previous work [46], we use the Azure serverless function traces [67, 90] as a proxy. Note the traffic in this trace is highly bursty and the model distribution is highly skewed. Except for the small-scale timeline analysis in Figure 16 (which we show for illustration purposes) and ablation studies, we run the traces for 5 minutes under different arrival rates and model distribution. Unless otherwise stated, there are 32 model variants that need to be served. The requests are sent to the serving system with a varying Poisson arrival rate (applied to the entire system).

Metrics. We use downstream accuracy to evaluate the post-compression quality and three metrics for serving performance: 1) end-to-end (E2E) latency averaged over all requests, 2) time to first token (TTFT) averaged over all requests, which is an important indicator of the system's responsiveness, 3) throughput and 4) SLO attainment (i.e., percentage of requests served within a given SLO requirement) for TTFT and E2E Latency.

Baselines. For compression quality, we compare DeltaZip with SparseGPT [29] which incorporates both quantization and sparsification, as well as AWQ [47] which is a state-of-the-art quantization algorithm. For serving performance, since there is no existing system that can serve multiple full-parameter fine-tuned models, we implement a simple baseline based on vLLM that supports 1) Swapping models in and out of GPU memory, 2) Continuous batching of different full-precision models by looping through models in a batch and 3) Batching available requests for the same model. We refer to this baseline as `vLLM-SCB`. We use the maximum number of models that can fit in the GPU memory for the baseline serving system. We use a tensor parallelism degree of 4 for both DeltaZip and the baseline system unless otherwise stated. We do not show a comparison to Serverless-LLM [32] as it treats each model as a black box — meaning it does not batch requests from models derived from the same base — and hence is not competitive for the scenarios we explore.

System Setup. We evaluate our serving system without tuning the maximal number of concurrent deltas, since it requires offline profiling and apriori knowledge of the workload. Instead, we show two representative configurations to reveal the system performance under different settings without explicit tuning.

6.2 Post-Compression Model Quality

We first study how inference accuracy is impacted by applying quantization and sparsification on model deltas compared to directly on FMT model weights. As shown in Table 1, DeltaZip achieves up to 13× compression ratio with comparable accuracy to the original FP16 FMT models. On smaller models, DeltaZip achieves 4× to 11× compression ratio with comparable accuracy. We also found the compression ratio for Gemma 2 models is lower than the Llama models. This is because the Gemma 2 models have a relatively larger number of parameters stored in the embedding

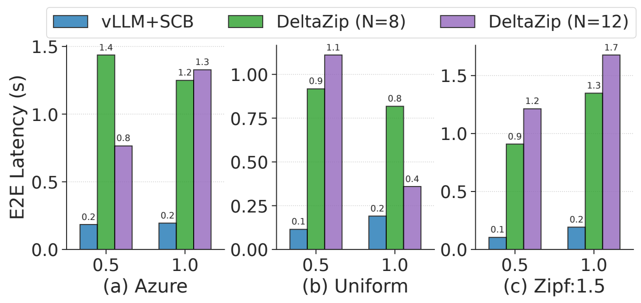
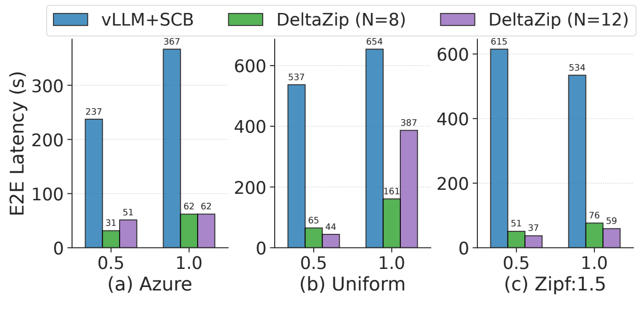
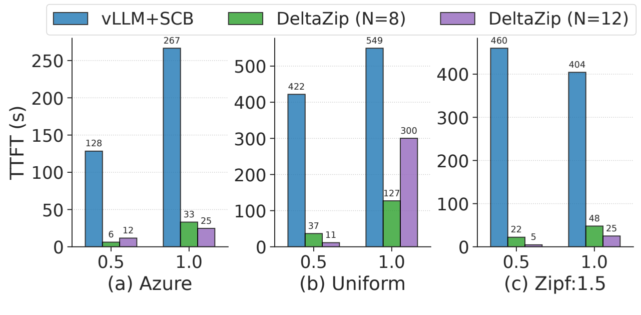


Figure 11. Throughput of different serving systems with varying poisson arrival rate $\lambda \in \{0.5, 1.0\}$ and distribution $\mathcal{D} \in \{\text{azure, uniform, zipf-1.5}\}$ for 13B model.



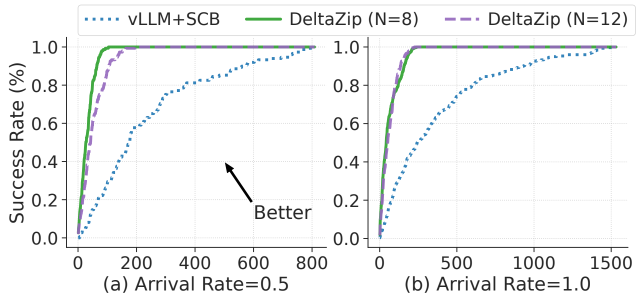
Average E2E Latency



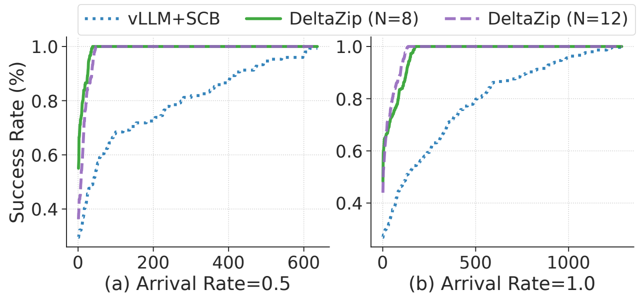
Average TTFT

Figure 12. Average latency of different serving systems with varying arrival rate and distribution for 13B model.

layers, which are not compressed. In contrast, SparseGPT — which applies similar techniques directly to the FMT model weights — has a substantial drop in accuracy for all models. DeltaZip also achieves comparable accuracy compared with the state-of-the-art quantization algorithm AWQ [47] but with a much higher compression ratio. We also observe that in some cases, the accuracy of the compressed model is even higher than the full-precision model. This effect is also noticeable in other works [31, 37, 47] and is likely due to the regularization effects of compression. These results show that ΔCompress can aggressively compress model deltas while retaining higher accuracy than the traditional approach of compressing model weights directly. In §6.4, we also show that compressed delta serving achieves higher accuracy on complex tasks than LoRA serving.



SLO attainment of E2E latency



SLO attainment of TTFT

Figure 13. SLO attainment of different serving systems with varying arrival rate and azure model distribution for 13B model. X-axis denotes different SLO requirement in seconds.

6.3 End-to-end Serving Performance

Serving throughput. Figure 11 shows the throughput of different serving systems with varying arrival rates and model distributions. We observe a 2× to 12× improvement in throughput compared to the baseline system. The improvement is more pronounced when the arrival rate is relatively light and more skewed. When the arrival rate is high and the model distribution is uniform, the improvement is less significant, and we find this is mainly due to the high cost in prompt processing when the models are more uniform. Since our techniques cannot reduce the prompt processing time, and when more deltas are batched together, requests in this batch have to wait for the prompt processing of the slowest request in the batch, leading to a lower throughput.

Average E2E latency and TTFT. Figure 12 shows that DeltaZip acheives a 1.6× to 16× improvement in average E2E latency and even higher improvement in TTFT. The high improvement in TTFT is due to DeltaZip's ability to batch more requests from different models concurrently, thus reducing queuing time. As in our throughput experiments, the improvement under uniform distribution with high load is less significant. We also observe that the maximum number of concurrent deltas being served has a substantial impact on the performance.

SLO Attainment of TTFT and E2E Latency. Figure 13 shows the SLO attainment of E2E latency and TTFT with the Azure trace under varying Poisson arrival rate. We observe

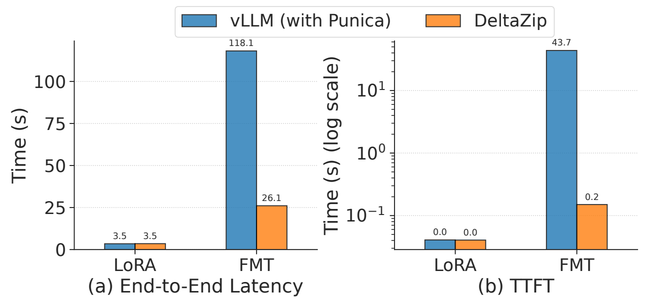


Figure 14. E2E latency and TTFT of DeltaZip and LoRA serving system when serving LoRA and FMT models.

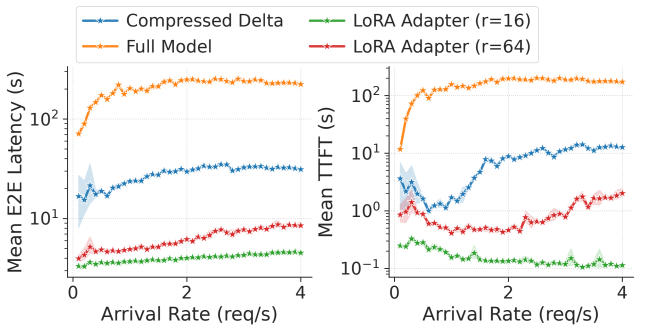


Figure 15. End-to-end latency and TTFT of DeltaZip and LoRA serving system with varying arrival rate.

Base Model	Task	Accuracy↑		
		FMT	LoRA	ΔCompress
Llama-7B	Math (GSM8K)	34.79	29.49	34.95
Pythia-2.8B	Amazon Review	73.76	50.92	73.13
	BoolQ Yes/No	79.61	63.76	79.52
	NLI Classification	73.23	52.15	70.74
OpenLlama 3B	Amazon Review	76.07	55.07	77.36
	BoolQ Yes/No	83.38	65.69	83.29
	NLI Classification	80.07	63.46	79.75

Table 2. Model quality of FMT vs. LoRA vs. ΔCompress.

that DeltaZip achieves a higher SLO attainment compared to the baseline system.

6.4 Delta versus LoRA Serving Approaches

Complementary to LoRA serving systems, DeltaZip extends their serving optimizations to also efficiently serve FMT models. DeltaZip inherits the ability to serve LoRA adapters from vLLM [43], which supports Punica-based [15] LoRA serving. Figure 14 shows an example of using DeltaZip to serve LoRA adapters on one GPU node and FMT model variants on another GPU node. For LoRA serving, DeltaZip achieves similar performance as vLLM with Punica kernels and for FMT serving, DeltaZip significantly outperforms the baseline due to its delta compression approach.

Given that DeltaZip supports both LoRA and FMT model serving, a natural question that arises for users is when to use which type of model fine-tuning approach. While a detailed study is outside the scope of this paper and prior works have already compared FMT and LoRA accuracy [8, 12], we conduct a brief analysis in which we apply delta compression for FMT serving. Table 2 compares the accuracy of uncompressed FMT models, LoRA adapters and ΔCompress compressed FMT models. We conduct extensive hyper-parameter tuning for LoRA adapters, using scripts from Anyscale [8]. We observe that even on complicated tasks (e.g., Math) where LoRA adapters cannot achieve similar accuracy as FMT models, ΔCompress can maintaining high accuracy while compressing the FMT models.

We also compare the inference latency and TTFT of LoRA adapter, compressed delta FMT and baseline full model FMT serving with varying arrival rates. Figure 15 shows that compressed deltas and LoRA adapters are much more efficient to serve than the vLLM+SCB baseline approach for FMT model serving, which swaps full models. Serving LoRA adapters is still more efficient than serving compressed deltas, mainly due to the more compact size and smaller memory footprint of LoRA adapters. In conclusion, **users should choose between LoRA and compressed delta FMT serving based on the trade-off between accuracy and serving performance**: LoRA is more suitable for tasks where accuracy is not the primary concern or for simpler tasks where LoRA can achieve comparable accuracy to full model tuning. In contrast, compressed delta FMT serving is more suitable for tasks where accuracy is critical. DeltaZip improves the serving efficiency for FMT serving with the ΔCompress delta compression approach and its optimized delta serving system design and implementation.

6.5 Microbenchmarks and Ablation Study

Latency breakdown. We perform a smaller-scale experiment to visualize the latency breakdown of different serving stages (queuing, loading, and inference). We synthesize a trace with 12 models, arrival rate of 0.5 requests per second for 60 seconds and run on two RTX 3090 GPUs. Figure 16 compares DeltaZip with the baseline. The vLLM+SCB system has two main issues: 1) the loading time is substantial, as it needs to load the entire model from the disk to GPU, and 2) queuing time dominates, due to the lack of batching. Even though it batches requests for the same model (e.g., for the model #2), it cannot batch requests from many different models due to GPU memory capacity limits. In contrast, DeltaZip alleviates these two issues by 1) loading only the compressed deltas, which can be 5× to 10× smaller than the full model, and 2) batching requests from different models, which significantly reduces queuing delays.

SBMM kernel. We evaluate the performance of the SBMM kernel described in §5.1. Figure 17 shows the performance of the SBMM kernel compared to the baseline implementations as we scale the number of models. We observe that the

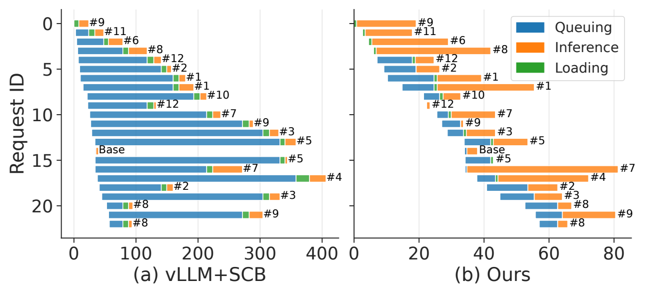


Figure 16. Serving latency breakdown. The # on the right indicates the model ID. The x-axis is the time in seconds.

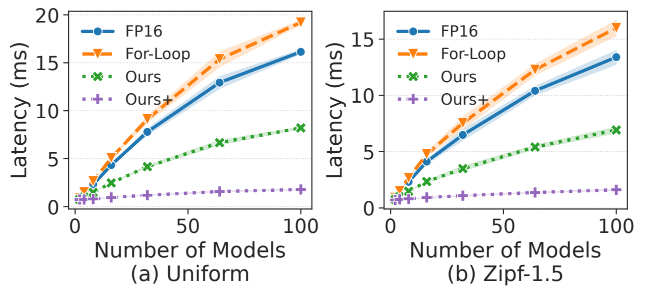


Figure 17. Microbenchmark of the SBMM kernel vs. the baseline implementation on a single GPU with varying number of models given a fixed number of requests. "Ours" refers to the implementation with reduced random memory access only and "Ours+" refers to the kernel implementation we proposed in §5.1. For FP16, we do not perform the decoupling as it will only introduce additional overhead.

naive for-loop approach does not bring much performance improvement and is comparable with the half-precision implementation. However, with reduced random memory access, we observe a 2× speedup compared to the baseline implementation, and the proposed kernel further improves the performance by another 2× to 3×. In addition, we also observe that given the same number of requests, our kernel scales well with different number of models.

Model Parallelism. We also conduct an experiment with varying numbers of GPUs under different settings to evaluate the model parallelism of DeltaZip. Figure 18 shows the end-to-end latency and TTFT of DeltaZip with varying number of GPUs. We observe that the latency decreases with the number of GPUs, particularly on A800 platform. This is because the inter-GPU communication is faster on A800 platform compared to the RTX 3090 platform. This observation leaves a direction for future work to optimize and tune the tensor parallelism degree of DeltaZip.

Starvation Handling. Next we evaluate the effectiveness of the preemption mechanism as described in §5.4. Figure 19 shows the E2E latency and TTFT with and without preemption. We observe that the preemption mechanism effectively reduces the latency, particularly the time-to-first-token as it allows more requests to start earlier.

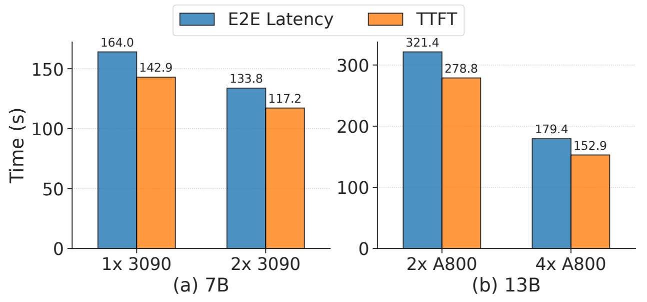


Figure 18. End-to-end latency and time to first token (TTFT) of DeltaZip with varying number of GPUs.

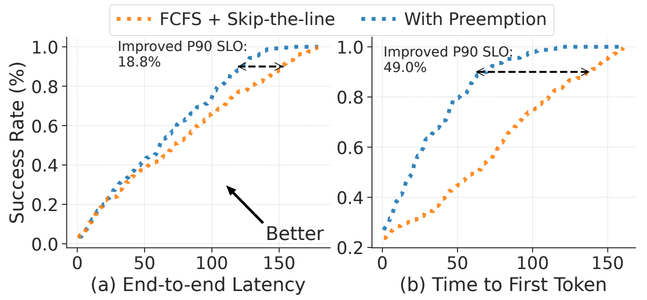


Figure 19. E2E Latency and TTFT of DeltaZip with and without preemption.

7 Related Work

ML Model Serving Systems. Optimizing ML serving is an active area of research [19, 36, 66]. With the increasing popularity of LLMs, there has been a surge of LLM serving systems proposing optimizations such as GPU kernel implementations [20, 21, 45, 86], advanced and fine-grained batching [26, 87], memory management [43], and parallelism [46, 69]. However, most of these works do not consider the multi-variant serving scenario, for which DeltaZip is designed. For multi-model LLM serving, MuxServe [25] explores spatial-temporal multiplexing for LLM serving to improve GPU utilization. ServerlessLLM [32] studies the feasibility of serving LLMs in a serverless environment and proposes fast checkpoint loading and locality-driven live migration. However, these works treat the models as a black box and do not consider their lineage, and are hence orthogonal to DeltaZip. Punica and S-LoRA [16, 68] are the most relevant to DeltaZip, as they also target multi-variant model serving scenarios, but they focus on serving LoRA adapters and do not consider serving full fine-tuned models. DeltaZip is complementary to these works.

Post-Training Compression. Another line of work focuses on reducing the memory footprint of large models by compressing the model weights in a lossy manner. Beyond the OBS framework [38] and its improvements, such as GPTQ [31] and SparseGPT [29], many compression algorithms [47, 76, 82, 85] have been proposed to reduce the model size. Such techniques can also be applied to model deltas by adapting the DeltaZip compression pipeline. There is also concurrent work on model delta compression. The initial version of our system [84] and GPT-Zip [41] proposes quantization and unstructured sparsity, but as this paper shows, with structured sparsity, we can achieve similar compression ratio and further improve the system performance. BitDelta [48] proposes extreme quantization and DARE [88] proposes unstructured sparsity. Compared to these works, DeltaZip optimizes more for **hardware efficiency** by combining quantization with *structured sparsity*. To the best of our knowledge, DeltaZip is also the first serving system to support both LoRA and compressed delta FMT model serving.

8 Discussion

Limitations. While DeltaZip's decoupled computation improves throughput and reduces latency when concurrently serving many models, decoupled inference still has higher unloaded latency than serving a FMT model directly in GPU memory. Hence, when there are only a handful of models to serve and they fit in GPU memory, DeltaZip may not be suitable. Compared to dedicated instances for each model, DeltaZip may be less performant, but it is more cost-effective and are more suited for environments where the cost and performance needs to be balanced. In addition, the co-serving of LoRA and FMT models in DeltaZip is at a coarse granularity, where LoRA and FMT models are served in two separate sets of GPUs and must be in separate batches. We plan to explore the possibility of serving LoRA and FMT models in the same batch as future work. Furthermore, DeltaZip reorders requests to to group requests belonging to the same delta together, thus it cannot guarantee the SLO constraints of individual models. Potential future work includes adding mechanisms to prioritize models based on their constraints, as well as predicting the output length of each request to better guarantee SLOs. Finally, the starvation handling mechanism in DeltaZip is simple and may not be beneficial for all cases. For example, the preemption of requests that are about to finish leads to unnecessary starvation and performance degradation. We plan to explore more sophisticated mechanisms, such as output length prediction, to handle starvation in future work.

Supporting PEFT approaches beyond LoRA. DeltaZip's decoupled computation architecture is general and can be used beyond LoRA, for other PEFT methods being proposed to improve accuracy. For example, GaLore [91] only uses low-rank on the gradient but results in full-rank weight updates. RoSA [53] introduces sparse adapters in addition to low-rank adapters. Due to the lack of support for full-rank weight updates, these methods cannot be served by existing LoRA-based systems. We plan to extend DeltaZip to add support for emerging PEFT methods.

9 Conclusion

To conclude, we propose DeltaZip, a serving system that enables LLM service providers to efficiently serve multiple fine-tuned models, whether they are fine-tuned through parameter-efficient or full-model tuning techniques. For efficient serving of full-model-tuned models, DeltaZip leverages a key insight: fine-tuning typically results in small perturbations, allowing model deltas to be highly compressible. DeltaZip co-designs the serving system with the compression algorithm and achieves 10× compression ratio, improves serving throughput by 2× to 12× and maintains high model quality comparable to FP16 models. One practical use case of DeltaZip is to pack less-popular models on a limited pool of GPUs, and for those popular models, service providers can still deploy them on dedicated machines, which allows for better resource allocation and minimizes operational costs.

Acknowledgments

We would like to thank Berivan Isik, Hermann Kumbong, Wanyi Ning, Ce Zhang, Sanmi Koyejo for their contribution and feedback during the early stage of the project. We also thank Lijie Xu, Yongjun He, Wenqi Jiang, Foteini Strati, anonymous EuroSys reviewers, and our shepherd, Thaleia Dimitra Doudali, for their insightful reviews and feedback. This work is partially supported by a Google Academic Research Award.

A Artifact Appendix

A.1 Abstract

A.2 Description & Requirements

Our open-source code for artifact evaluation can be found at: `https://github.com/eth-easl/deltazip-ae` as well as https://doi.org/10.5281/zenodo.14870277. We also provide a docker image (`ghcr.io/xiaozheyao/deltazip`) for easy setup of the software environment. The README file in the repository provides instructions on how to use the code and produce results.

A.2.1 How to access Our code is available at: https://github.com/eth-easl/deltazip-ae. The docker image is available at: `ghcr.io/xiaozheyao/deltazip`.

A.2.2 Hardware dependencies For the compression experiments, our reproduce script requires a machine with NVIDIA ampere GPUs (e.g., A100, RTX 3090, etc.) with equal or more than 24GB GPU memory. The machine should have at least 100 GB of storage to store the model weights and compressed models.

For the performance experiments, our reproduce script requires a machine with 4 NVIDIA A100 GPUs, and over 1TB of storage. The machine should have at least 480GB of RAM.

A.2.3 Software dependencies We provide a docker image for easy setup of the software environment. In order to run the docker image, the host machine should have `Docker` and `nvidia-container-toolkit` installed. The host machine should also have installed `CUDA 12.0` and above.

If you prefer to run the code without the docker image, the software dependencies can be found in the `requirements.txt` file under `compression` and `serving` directories. We recommend refer to the `Dockerfile` for the exact dependencies of our artifact.

Notably, we also rely on lm-eval-harness to evaluate the accuracy of the models.

A.2.4 Benchmarks

A.3 Set-up

Please set up the environment variables and paths using `scripts/env.sh`.

A.4 Evaluation workflow

Our evaluation workflow consists of two parts: compression/model quality evaluation and performance evaluation.

A.4.1 Major Claims Our major claims are that the model deltas can be compressed by a factor of `10x` into 2-bit per parameter and 50% structured sparsity, with minimal loss in model quality. Such compression can be leveraged to improve the performance of the model serving system.

A.4.2 Experiments Experiment (E1): [Compressing 7B Model] [30 human-minutes + 3 compute-hours]: We compress the 7B model to 2-bit per parameter and 50% structured sparsity. We expect the compressed model to have a compression ratio of 10x with minimal loss in model quality.

```
bash scripts/1_compress_7b.sh
```

The above command should take around 30 minutes to run.

We then test that the model can generate reasonable text samples.

```
bash scripts/1.1_test_generate.sh
```

We then evalaute the model quality using the lm-eval-harness toolkit, by running the following command:

```
bash scripts/2_eval_quality_compressed.sh
```

The above commands will print the accuracy of the model in a table format.

To compare the model quality of the compressed model with the original model, we evaluate the performance of the original model using the following command:

```
bash scripts/2_eval_quality_original.sh
```

To compare the model quality with other baseline compression techniques, we first compress the models using SparseGPT and AWQ, by running the following commands:

```
bash scripts/2.1_compress_baselines.sh
```

Then we can evaluate the model quality of the compressed models using the following command:

```
bash scripts/2.2_eval_baselines.sh
```

We provide a helper script to summarize the results:

```
python scripts/helpers/aggregate_accuracy.py
    --compressed-model $WORKDIR/
    compressed_models/lmsys.vicuna-7b-v1.5.2
    b_2n4m_128bs --full-model lmsys/vicuna-7b-
    v1.5 --accuracy-dir $WORKDIR/eval_results
    --sparsegpt-model $WORKDIR/
    sparsegpt_models/lmsys.vicuna-7b-v1.5.4
    b_2n4m_128bs --awq-model $WORKDIR/
    awq_models/awq.lmsys.vicuna-7b-v1.5.4b128g
```

Experiment (E2): [Performance Evaluation] [60 human-minutes + 5 compute-hours]:

In order to evaluate the serving performance of our system, we first need to compress the 13b model, prepare the delta weights and create multiple copies.

```
bash scripts/3_compress_13b.sh
bash scripts/3_prepare_delta.sh
```

We pre-partition the model weights into the tensor-parallel shards. If you change the tensor-parallel degree, you need to re-partition the model weights. By default, we set tensor parallel degree to be 4.

In another terminal, run the following command to start the client:

```
python3 scripts/helpers/bench.py --workload
    scripts/workload/azure.ar=0.5.jsonl --base
    -model meta-llama/Llama-2-13b-hf --output
    $WORKDIR/results
```

Note that the performance numbers may vary depending on the exact machine you are running the evaluation on. In general, we expect DeltaZip to have lower latency and higher throughput compared to the baseline.

References

[1] Numbers every llm developer should know. https://github.com/ray-project/llm-numbers, 2023.

[2] Nvlink and nvswitch. https://www.nvidia.com/en-us/data-center/nvlink/, 2023.

[3] Meta llama 3. https://github.com/meta-llama/llama3, 2024.

[4] Alpaca: A strong, replicable instruction-following model. https://crfm.stanford.edu/2023/03/13/alpaca.html/, 2025.

[5] Amd. https://rocm.docs.amd.com/projects/hipSPARSELt/en/docs-5.7.0/API_Reference_Guide.html/, 2025.

[6] nomic-ai/gpt4all-13b-snoozy model card. https://huggingface.co/nomic-ai/gpt4all-13b-snoozy/, 2025.

[7] Adinets Andy. CUDA Dynamic Parallelism API and Principles, May 2014.

[8] Anyscale. Fine-Tuning LLMs: LoRA or Full-Parameter? An in-depth Analysis with Llama 2, November 2023.

[9] Rob Armstrong, Rob Nertney, Arthy Sundaram, Matthew Nicely, Robert Jensen, and Fred Oh. Cuda toolkit 12.0 released for general availability. https://developer.nvidia.com/blog/cuda-toolkit-12-0-released-for-general-availability/, 2022.

[10] b mc2. sql-create-context dataset, 2023.

[11] Hongxiao Bai and Yun Li. Structured Sparsity in the NVIDIA Ampere Architecture and Applications in Search Engines, July 2023.

[12] Dan Biderman, Jose Gonzalez Ortiz, Jacob Portes, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley,

Jonathan Frankle, Cody Blakeney, and John P. Cunningham. Lora learns less and forgets less, 2024.

[13] Roberto L. Castro and Dan Alistarh. Sparse-marlin: Boosting 4-bit inference kernels with 2:4 sparsity. https://github.com/IST-DASLab/Sparse-Marlin, 2024.

[14] Yuji Chai, John Gkountouras, Glenn G. Ko, David Brooks, and Gu-Yeon Wei. INT2.1: Towards Fine-Tunable Quantized Large Language Models with Error Correction through Low-Rank Adaptation, June 2023. arXiv:2306.08162 [cs].

[15] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-Tenant LoRA Serving, October 2023. arXiv:2310.18547 [cs].

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

[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. arXiv:2107.03374 [cs].

[18] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[19] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} Online Prediction Serving System. pages 613–627, 2017.

[20] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning, July 2023. arXiv:2307.08691 [cs].

[21] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, June 2022. arXiv:2205.14135 [cs].

[22] DeepInfra. Deepinfra custom llms inference api. https://deepinfra.com/pricing, 2025.

[23] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations, 2023.

[24] Thomas Dohmke. Github copilot x: The ai-powered developer experience, March 2023.

[25] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible Multiplexing for Efficient Multiple LLM Serving, April 2024. arXiv:2404.02015 [cs].

[26] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System For Transformer Models, February 2021. arXiv:2010.05680 [cs].

[27] Fireworks.AI. Fireworks on-demand deployments. https://fireworks.ai/pricing, 2025.

[28] PyTorch Foundation. torch.bmm — PyTorch 2.3 documentation.

[29] Elias Frantar and Dan Alistarh. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot, March 2023. arXiv:2301.00774 [cs].

[30] Elias Frantar and Dan Alistarh. Marlin: a fast 4-bit inference kernel for medium batchsizes. https://github.com/IST-DASLab/marlin, 2024.

[31] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers, March 2023. arXiv:2210.17323 [cs].

[32] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Locality-enhanced serverless inference for large language models. *arXiv preprint arXiv:2401.14351*, 2024.

[33] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023.

[34] Xinyang Geng, Arnav Gudibande, Hao Liu, Eric Wallace, Pieter Abbeel, Sergey Levine, and Dawn Song. Koala: A dialogue model for academic research. Blog post, April 2023.

[35] GitHub. GitHub Copilot · Your AI pair programmer.

[36] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. pages 443–462, 2020.

[37] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[38] B. Hassibi, D.G. Stork, and G.J. Wolff. Optimal Brain Surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pages 293–299 vol.1, March 1993.

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

[40] Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. Accelerated Sparse Neural Training: A Provable and Efficient Method to Find N:M Transposable Masks. In *Advances in Neural Information Processing Systems*, volume 34, pages 21099–21111. Curran Associates, Inc., 2021.

[41] Berivan Isik, Hermann Kumbong, Wanyi Ning, Xiaozhe Yao, Sanmi Koyejo, and Ce Zhang. GPT-Zip: Deep Compression of Finetuned Large Language Models. July 2023.

[42] Juraj Juraska, Kevin K. Bowden, and Marilyn Walker. ViGGO: A Video Game Corpus for Data-To-Text Generation in Open-Domain Conversation, October 2019. arXiv:1910.12129 [cs].

[43] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 611–626, New York, NY, USA, October 2023. Association for Computing Machinery.

[44] Yann LeCun, John Denker, and Sara Solla. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[45] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[46] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. {AlpaServe}: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. pages 663–679, 2023.

[47] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song

Han. Awq: Activation-aware weight quantization for llm compression and acceleration. In *Proceedings of Machine Learning and Systems*, MLSys '24, 2024.

[48] James Liu, Guangxuan Xiao, Kai Li, Jason D Lee, Song Han, Tri Dao, and Tianle Cai. Bitdelta: Your fine-tune may only be worth one bit. *arXiv preprint arXiv:2402.10193*, 2024.

[49] Peiyu Liu, Zikang Liu, Ze-Feng Gao, Dawei Gao, Wayne Xin Zhao, Yaliang Li, Bolin Ding, and Ji-Rong Wen. Do Emergent Abilities Exist in Quantized Large Language Models: An Empirical Study, July 2023. arXiv:2307.08072 [cs].

[50] Tiedong Liu and Bryan Kian Hsiang Low. Goat: Fine-tuned LLaMA Outperforms GPT-4 on Arithmetic Tasks, May 2023. arXiv:2305.14201 [cs].

[51] madiepev. Fine-tune a foundation model from the model catalog in Azure Machine Learning - Training.

[52] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-Task Generalization via Natural Language Crowdsourcing Instructions, March 2022. arXiv:2104.08773 [cs].

[53] Mahdi Nikdan, Soroush Tabesh, Elvir Crnčević, and Dan Alistarh. RoSA: Accurate Parameter-Efficient Fine-Tuning via Robust Adaptation, June 2024. arXiv:2401.04679 [cs].

[54] NVIDIA. NVCOMP, March 2021.

[55] OpenAI. GPT-3.5 Turbo fine-tuning and API updates.

[56] OpenAI. Introducing ChatGPT.

[57] OpenAI. OpenAI Pricing.

[58] OpenAI. GPT-4 Technical Report, March 2023. arXiv:2303.08774 [cs].

[59] OpenRouter. Fireworks on-demand deployments. https://openrouter.ai/, 2025.

[60] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, March 2022. arXiv:2203.02155 [cs].

[61] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting, May 2024. arXiv:2311.18677 [cs].

[62] Google Cloud Platform. Vertex AI.

[63] PrediBase. Predibase serverless solution for fine-tuned llms. https://predibase.com/blog/introducing-the-first-purely-serverless-solution-for-fine-tuned-llms, 2025.

[64] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model, December 2023. arXiv:2305.18290 [cs].

[65] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, September 2023. arXiv:1910.10683 [cs, stat].

[66] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated Model-less Inference Serving. pages 397–411, 2021.

[67] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. pages 205–218, 2020.

[68] 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. S-LoRA: Serving Thousands of Concurrent LoRA Adapters, November 2023. arXiv:2311.03285 [cs].

[69] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, March 2020. arXiv:1909.08053 [cs].

[70] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. D\'ej\`aVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving, March 2024. arXiv:2403.01876 [cs].

[71] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[72] Gemini Team. Gemini: A Family of Highly Capable Multimodal Models, May 2024. arXiv:2312.11805 [cs].

[73] TogetherAI. Together.ai custom models inference api. https://www.together.ai/products#inference, 2025.

[74] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, February 2023. arXiv:2302.13971 [cs] version: 1.

[75] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. *arXiv preprint arXiv:2402.04396*, 2024.

[76] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: Even Better LLM Quantization with Hadamard Incoherence and Lattice Codebooks, June 2024. arXiv:2402.04396 [cs].

[77] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[78] Panqi (William) Wei. PanQiWei/AutoGPTQ, December 2023. original-date: 2023-04-13T02:18:11Z.

[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-Art Natural Language Processing. In Qun Liu and David Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

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

[81] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[82] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models, June 2023. arXiv:2211.10438 [cs].

[83] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.

[84] Xiaozhe Yao and Ana Klimovic. Deltazip: Multi-tenant language model serving via delta compression. *arXiv preprint arXiv:2312.05215*, 2023.

[85] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. ZeroQuant: Efficient and Affordable

Post-Training Quantization for Large-Scale Transformers, June 2022. arXiv:2206.01861 [cs].

[86] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. Accelerating self-attentions for llm serving with flash-infer, February 2024.

[87] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for {Transformer-Based} Generative Models. pages 521–538, 2022.

[88] Le Yu, Bowen Yu, Haiyang Yu, Fei Huang, and Yongbin Li. Language models are super mario: Absorbing abilities from homologous models as a free lunch. In *Forty-first International Conference on Machine Learning*, 2024.

[89] Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method, February 2024. arXiv:2402.17193 [cs].

[90] Yanqi Zhang, Inigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 724–739, New York, NY, USA, October 2021. Association for Computing Machinery.

[91] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

[92] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena, July 2023. arXiv:2306.05685 [cs].

[93] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena, July 2023. arXiv:2306.05685 [cs].

[94] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving, June 2024. arXiv:2401.09670 [cs].

[95] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch, April 2021. arXiv:2102.04010 [cs].

[96] Xuanhe Zhou, Guoliang Li, and Zhiyuan Liu. LLM As DBA, August 2023. arXiv:2308.05481 [cs].

[97] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. {PetS}: A Unified Framework for {Parameter-Efficient} Transformers Serving. pages 489–504, 2022.