IMPLEMENTING MULTI-TENANT LORA SERVING ON MINITORCH

Zengliang Zhu [*1] **Xinran Wan** [*1] **Kath Choi** [*1]

ABSTRACT

LoRA enhances model fine-tuning by adding decomposition matrices to the original weight matrix, optimizing for specific tasks across multiple models. This work explore serving multi-tenant LoRA using the innovative Segmented Gather Matrix-Vector Multiplication (SGMV) CUDA kernel, introduced by the Punica framework. Our methodology, inspired by Chen et al. (2023), employs a single NVIDIA GPU setup to examine the LoRA batching effect on a customized DecoderLM model. We aim to assess the computational efficiency and performance when serving multi-tenant LoRA through the Punica framework. The SGMV kernel, which facilitates segmented operations for matrix-vector multiplication, is central to our implementation, expected to showcase significant performance improvements. This work seeks to underscore Punica's potential in enhancing computational resource utilization for multi-tenant LoRA serving and inference efficacy across diverse LLM applications. Our implementation can be found in https://github.com/bchuh/miniMultiLora which is currently a private repository.

1 INTRODUCTION

LoRA (Low-Rank Adaptation) is an efficient fine-tuning technique that incorporates trainable matrix decomposition into the linear layers of transformer models. It has gained popularity for fine-tuning large transformers across various applications, including large language models (LLMs), vision models, speech representation models, and diffusion models. Remarkably, LoRA reduces memory usage by up to five times(Lester et al., 2021) during the fine-tuning phase while delivering performance comparable to traditional fine-tuning methods. LoRA finetuning can not only alter the writing style, and capability of LLMs, lately, it has also been proposed that LoRA can be used to store long-term memory of conversations with the user(Wang et al., 2024).

As the demand for customized LLMs increases, many providers have begun offering fine-tuning services. In these services, clients supply data, and the providers manage the fine-tuning and deployment. This approach, known as centralized customization, consolidates the hosting of various fine-tuned models for different clients onto a single platform. This method not only optimizes cluster utilization, reducing costs, but also simplifies the complexity associated with model fine-tuning. However, it raises an important question about how to efficiently serve a vast array of customized LLMs simultaneously.

Typically, serving a LoRA-tuned LLM involves merging the learned decomposition matrix with the model's weights, enabling adaptation without additional computation during the inference phase. Yet, deploying numerous LoRA-tuned LLMs simultaneously introduces redundancy, as it requires loading a complete copy of the LLM for each client. LoRA's methodology can be described by the following formulas, where ADT denotes the LoRA adapter, ATT represents the attention layer, and h indicates the dimensions of the Key and Value matrices:

$$H_i = \text{ATT}\left(XW_q^{(i)}, \text{ADT}_k(X) + XW_k^{(i)}, \text{ADT}v(X) + XW^{(i)}v\right) \quad (1)$$

$$\text{ADT}(X) = XWd_h \times d_mWd_m \times d_h \quad (2)$$

Figure 1. Illustration of current multi-LoRA and Punica multi-LoRA serving methods

[*]Equal contribution [1]Language Technology Institution, Carnegie Mellon University, Pennsylvania, USA. Correspondence to: Zengliang Zhu <zengliaz@andrew.cmu.edu>, Xinran Wan <xwan2@andrew.cmu.edu>, Kath Choi <yeemanc@andrew.cmu.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2024.

This approach reveals that although each input batch may request different ADT adapters, the computation for the base model can be shared across requests. Consequently, only a single copy of the base model needs to be loaded, and a collection of customized LoRA adapters is required for adapter-specific computations. This strategy optimizes resource use and facilitates the efficient serving of multiple customized models concurrently.

In this project, we adopt the Punica framework and implement the customized CUDA kernels essential to unleash the batch processing capability when serving different LoRA. We then bind the customized CUDA kernels to minitorch and provide a customized DecoderLM, the result demonstrated 4x performance improvement with 4x less number of parameters and little overhead.

The major contribution of this work is:

1. Implementation of the customized CUDA kernels to enable batch processing requests for different LoRA
2. Binding CUDA kernels to minitorch and adopt the kernel in customized DecoderLM
3. Analyse the effectiveness and performance improvement when adopting the Punica framework

2 Background and Related Work

We will look at some background work relevant to the tasks of Punica on transfomer-based LLMs and LoRA. We also provide context on LLM inference system optimization and multi-model inference serving.

2.1 Transformer-based LLMs

Transformer architecture is first introduced by Vaswani et al. (2023), and is widely used in current LLMs. Transformer-based LLMs take the users' prompt and generate a sequence of tokens, and also the Key-Value cache (KvCache). In the iterative decoding stage, a token and the KvCache are used to generate a new token and append a new column in the KvCache. Each Transformer block consists of a self-attention layer and a multi-layer perceptron (MLP). The prefill stage of transformer-based LLMs can fully utilize the GPU with latency proportional to the batch size. Since the input to the decoding stage is a single vector, there is low GPU utilization. Yu et al. (2022) showed that batching is an effective method to improve GPU utilization for the decoding step, which is essential to the serving latency and throughput of Transfomer-based LLMs.

2.2 Inference Systems

2.2.1 *LLM Inference Optimization*

Growing demands for ML-relevant applications makes the ML inference serving system a critical component in the backend. The typical workflow of such a system starts with the end-user submitting requests to an inference service, and the service replies based on a pre-defined ML model using its resources such as GPUs and TPUs. A number of recent work already investigated into the field of optimizing LLM inference, Orca(Yu et al., 2022) splits concatenated batch input at the self-attention operation to batch transformer-based text generation, vLLM(Kwon et al., 2023) minimizes the fragmentation of KvCache memory by adopting the concept of virtual pages from operating systems. FlashAttention(Dao et al., 2022) offers an enhanced implementation of the self-attention operation by decreasing data transfers through block-wise computation.

2.2.2 *Multi-model Inference Serving*

Most of previous works on serving ML models on GPU cluster focused on small, stateless models such as CNN(Gujarati et al., 2020), but serving LLM requires KvCache that persists states. Other serving frameworks considered different designs but limitations persist as compared to Punica. Some system allows GPU-sharing of the pre-trained models for different downstream tasks(Zhou et al., 2022), but they fail to run different adapted models concurrently.

2.3 Low-Rank Adaptation (LoRA)

LoRA(Hu et al., 2021), standing for Low-Rank Adaptation, emerges as a cutting-edge fine-tuning methodology that integrates trainable matrix decomposition within the linear layers of transformer-based models. This technique has been widely adopted for refining large transformers utilized in a broad spectrum of applications, ranging from large language models (LLMs) and vision models to speech representation and diffusion models. One of LoRA's most significant contributions is its ability to drastically reduce memory usage during the fine-tuning phase—by up to five times—without compromising on the performance levels achieved through traditional fine-tuning methods.

LoRA's fine-tuning capability extends beyond mere performance enhancements; it allows for modifications in the writing style and capabilities of LLMs. Recent advancements suggest LoRA's potential in embedding long-term memory of user interactions into models, paving the way for more personalized applications.

2.3.1 *Serving multi-tenant LoRA*

Serving LoRa could be done by serving LoRa as an independent model with existing LLM serving systems such as vLLM. However, these approaches are not optimized for the serving of multiple LoRA in a shared GPU cluster. Prior to the publication of this paper, there was limited work on multi-LoRA serving systems. This paper explored the opportunities to serve multiple LoRa on a cluster using

only one copy of the pre-trained model, and significantly improved the memory and computation utilization of GPU. Punica is capable of achieving higher throughput with negligible latency overhead when serving multi-tenant LoRA, enabling an efficient way to serve various LoRA at once.

3 METHOD

The core of Punica is the design of a new CUDA kernel by which the authors called it Segmented Gather Matrix-Vector Multiplication (SGMV). We know from previous section that LoRA fintunes pre-trained LLMs by adding two smaller matrices A and B. Hence given an input x and initial pre-trained wight matrix W, we can calculate the output:

$$y := \mathbf{x} \cdot (W + A \cdot B)$$

One LoRA model is used for one specific task. Hence if there are n LoRA models and $2n$ associated matrics $A_1, B_1, \ldots, A_n, B_n$, and input x is defined as $\mathbf{x} := (\mathbf{x_1}, \ldots, \mathbf{x_n})$, the equation becomes:

$$\begin{aligned} y &:= \mathbf{x} \cdot (W + A \cdot B) \\ &:= \mathbf{x} \cdot W + \mathbf{x} \cdot A \cdot B \\ &:= \mathbf{x} \cdot W + \mathbf{x_1} \cdot A_1 \cdot B_1 \cdots + \mathbf{x_n} \cdot A_n \cdot B_n \end{aligned}$$

Calculating $x@W$ is fast because it's merely inputting x to the original pre-trained model weights, and SGMV is used to make the calculation of $\mathbf{x} \cdot W + \mathbf{x_1} \cdot A_1 \cdot B_1 \cdots + \mathbf{x_n} \cdot A_n \cdot B_n$ more efficient.

Figure 3 illustrates how the input is further batched and calculated by SGMV. The SGMV operator consists of two parts: SGMV-shrink and SGMV-expand, the former takes care of calculating $v := x \cdot A$, and the latter calculates $y := v \cdot B$. For both kernels, the LoRA index is bound to BLOCKIDX.Y in CUDA. Further details involving different schedules for matrix multiplication under expand and shrink settings are illustrated in the original paper, which we will follow in our implementation. (Chen et al., 2023) There are several differences between our implementation and the one described in the paper. First, we bind LoRA index to BLOCKIDX.Z instead of BLOCKIDX.Y, as BLOCKIDX.Y is already used for matrix multiplication. Second, the origin paper partitioned the narrow matrix A and B into chunks and assign each chunk to a thread block. In our implementation, we used tiling, which provides the same level of parallelism. The two phases, SGMV-shrink and SGMV-expand, reuses one SGMV kernel, which we will further discuss in the next section.

3.1 CUDA Implementation

The following is our implementation of the SGMV CUDA kernel. The `SGMVKernel` is designed to perform a segmented gather matrix-vector multiplication (SGMV) operation in parallel on a GPU. It is particularly tailored for cases where different segments of an input matrix need to be processed with different transformations, specifically different sets of LoRA weights. The kernel initiates by identifying the number of LoRA models (n_{lora}), which corresponds to the depth of the CUDA grid in the z-dimension (`blockDim.z`). It computes an offset (batch_size_offset) based on an index array (`lora_idx_s`), indicating the beginning of each segment, facilitating the processing of inputs where different segments are handled by distinct LoRA models. Two shared memory arrays, `M1_shared` and `M2_shared`, store tiles of the matrices $M1$ and $M2$. Note that $M1$ and $M2$ means the two input matrices in the matrix multiplication. This common technique in CUDA for matrix multiplication reduces the need for frequent global memory accesses. Each block of threads manages a specific "batch" of the operation, with each thread responsible for computing elements of the resulting matrix. The indices i and j determine the row and column of the output matrix that the current thread will compute.

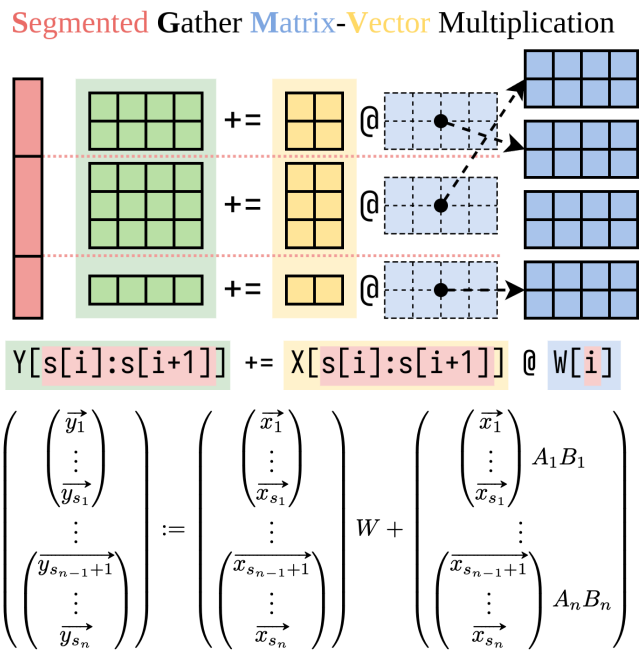

Figure 2. Illustration of Segmented Gather Matrix-Vector Multiplication (SGMV)

During the kernel execution, it iterates over tiles of the input matrices. For each tile, it loads the appropriate elements of matrices $M1$ and $M2$ into shared memory. If indices are outside the valid range (as defined by row_limit and matrix dimensions), zeros are loaded to prevent invalid memory accesses. Once the shared memory is populated, each thread

computes a partial sum for the output matrix element it is responsible for by multiplying corresponding elements from the loaded tiles of $M1$ and $M2$. After all tiles have been processed, each thread writes its computed value to the output matrix at the calculated position, ensuring that it only writes within the bounds specified by row_limit.

The function `launchSGMV` sets up and launches the `SGMVKernel`. It handles memory allocation, data transfer between the host and the device, and configuration of kernel launch parameters. It allocates GPU memory for the input and output matrices as well as for A, B, and additional necessary arrays like shapes and strides. Data is then copied from the host to the device, preparing everything needed for the kernel execution. The kernel configuration and launch involve calculating dimensions for thread blocks and grid based on the maximum size of LoRA groups and other parameters. The `SGMVKernel` is then launched twice—once for computing $v := x \cdot A$ and once for computing $y := v \cdot B$, corresponding to the SGMV-shrink and SGMV-expand operations, respectively. After the kernel executions, it copies the results back to the host and frees up the allocated GPU memory.

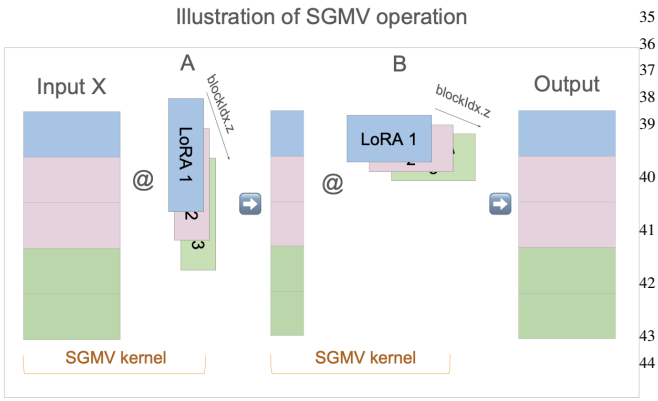

Figure 3. SGMV Kernel Implementation Overview

```
__global__ void SGMVKernel(
  scalar_t* out,
  const int* out_shape,
  const int* out_strides,
  scalar_t* a_storage,
  const int* a_shape,
  const int* a_strides,
  scalar_t* b_storage,
  const int* b_shape,
  const int* b_strides,
  const int* lora_idx_s
) {
  // general SGMV kernal, call twice with different a and b can work as expand or shrink kernal
  int n_lora = blockDim.z;
  int batch_size_offset = lora_idx_s[blockIdx.z];
  int row_limit[2]; //dim order: (row, col). row_limit = {lower_bound_inclusive, upper_bound_exclusive}
  // this is the row idx limit of the output matrix
  row_limit[0] = lora_idx_s[blockIdx.z]; // inclusive. lora_idx_s[blockIdx.z] is the start of current group
  row_limit[1] = lora_idx_s[blockIdx.z+1]; // exclusive


  ///begin mat mult: a*b ///


  __shared__ scalar_t a_shared[TILE][TILE];
  __shared__ scalar_t b_shared[TILE][TILE];

  // In each block, we will compute a batch of the output matrix
  // All the threads in the block will work together to compute this batch
  int batch = blockIdx.z;

  /// TODO
  // Hints:
  int temp_pos_2d[2];
  int temp_pos_3d[3];
  //int int_index[MAX_DIMS];

  // 1. Compute the row and column of the output matrix this block will compute
  int i = batch_size_offset + blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  // 2. Compute the position in the output array that this thread will write to

  int N = a_shape[2]; //N here means m, the second dim of input x

  // 3. Iterate over tiles of the two input matrices, read the data into shared memory
  int tile_i = threadIdx.x;
  int tile_j = threadIdx.y;
  double out_temp = 0;
  for(int ks = 0; ks < N; ks+=TILE){
      temp_pos_2d[0] = i;
      temp_pos_2d[1] = ks+tile_j;
      if(i<row_limit[1] && temp_pos_2d[1] < N){
          a_shared[tile_i][tile_j] = a_storage[index_to_position(temp_pos_2d, a_strides, 2)];
      }else
          a_shared[tile_i][tile_j] = 0;
      temp_pos_3d[0] = blockIdx.z;
      temp_pos_3d[1] = ks+tile_i;
      temp_pos_3d[2] = j;
      if (temp_pos_3d[1] < N && j < b_shape[2]){
```

```
            b_shared[tile_i][tile_j] =
                b_storage[index_to_position(
                temp_pos_3d, b_strides, 3)];
        }else
            b_shared[tile_i][tile_j] = 0;

        __syncthreads();
        for(int ki = 0; ki < TILE; ki++){
            out_temp += a_shared[tile_i][ki]
                * b_shared[ki][tile_j];
        }

        __syncthreads();
    }
    if (i<row_limit[1] && j<out_shape[1]){
        temp_pos_2d[0] = i;
        temp_pos_2d[1] = j;
        out[index_to_position(temp_pos_2d,
            out_strides, 2)] = out_temp;

    }
}
```

Listing 1. SGMV kernel

```
void launchSGMV(
    float* in_storage,
    int* in_shape,
    int* in_strides,
    float* out,
    int* out_shape,
    int* out_strides,
    float* a_storage,
    int* a_shape,
    int* a_strides,
    float* b_storage,
    int* b_shape,
    int* b_strides,
    int* lora_idx_s,
    int m, int p
) {
    // in matrix should be batch_size * m
    // a and b (lora matrixs) should be
        n_lora * m * n and n_lora * n * p
    // m and p here means the in_dim and
        out_dim of actural linear weight
        matrix
    // lora_idx_s: array of idxs, 0<=
        lora_idx_s[i]<batch_size, represent
         the start idx of each lora input
        group.
    // e.g., input 0~7: lora A, input 8~10:
         lora B. lora_idx_s: [0, 7, 10]
    // m: input hidden_dim
    int batch = in_shape[0];
    int n_lora = a_shape[0];
    int n = a_shape[2];
    // max_lora_group_size: the size (
        number of tokens) of the largest
        lora group in input matrix
    int max_lora_group_size =
        maxLoraGroupSize(lora_idx_s, n_lora
        +1);
    // n means lora rank (low rank space)
    // Allocate device memory
    float *d_out, *d_a, *d_v, *d_b, *d_in;
    cudaMalloc(&d_in, batch * m * sizeof(
        float));
    cudaMalloc(&d_a, n_lora * m * n *
        sizeof(float));
    cudaMalloc(&d_b, n_lora * n * p *
        sizeof(float));
    cudaMalloc(&d_v, batch * n * sizeof(
        float));
    cudaMalloc(&d_out, batch * p * sizeof(
        float));

    int *d_out_shape, *d_out_strides, *
        d_a_shape, *d_a_strides, *d_b_shape
        , *d_b_strides, *d_in_shape, *
        d_in_strides, *d_lora_idx_s, *
        d_v_shape, *d_v_strides;
    cudaMalloc(&d_in_shape, 2 * sizeof(int)
        );
    cudaMalloc(&d_in_strides, 2 * sizeof(
        int));
    cudaMalloc(&d_out_shape, 2 * sizeof(int
        ));
    cudaMalloc(&d_out_strides, 2 * sizeof(
        int));
    cudaMalloc(&d_a_shape, 3 * sizeof(int))
        ;
    cudaMalloc(&d_a_strides, 3 * sizeof(int
        ));
    cudaMalloc(&d_b_shape, 3 * sizeof(int))
        ; // leave a, b as 3d because first
         dim is n_lora
    cudaMalloc(&d_b_strides, 3 * sizeof(int
        ));
    cudaMalloc(&d_lora_idx_s, (n_lora+1) *
        sizeof(int)); //+1 because the
        first element is 0
    cudaMalloc(&d_v_shape, 2 * sizeof(int))
        ;
    cudaMalloc(&d_v_strides, 2 * sizeof(int
        ));
    int v_shape[2] = {batch, n};
    int v_strides[2] = {n, 1};
    // Copy data to the device
    cudaMemcpy(d_in, in_storage, batch * m
        * sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_a, a_storage, n_lora * m *
         n * sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b_storage, n_lora * n *
         p * sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_in_shape, in_shape, 2 *
        sizeof(int), cudaMemcpyHostToDevice
        );
    cudaMemcpy(d_in_strides, in_strides, 2
        * sizeof(int),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_out_shape, out_shape, 2 *
        sizeof(int), cudaMemcpyHostToDevice
        );
    cudaMemcpy(d_out_strides, out_strides,
        2 * sizeof(int),
```

```
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_a_shape, a_shape, 3 *
        sizeof(int), cudaMemcpyHostToDevice
        );
    cudaMemcpy(d_a_strides, a_strides, 3 *
        sizeof(int), cudaMemcpyHostToDevice
        );
    cudaMemcpy(d_b_shape, b_shape, 3 *
        sizeof(int), cudaMemcpyHostToDevice
        );
    cudaMemcpy(d_b_strides, b_strides, 3 *
        sizeof(int), cudaMemcpyHostToDevice
        );
    cudaMemcpy(d_lora_idx_s, lora_idx_s, (
        n_lora+1) * sizeof(int),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_v_shape, &v_shape, 2 *
        sizeof(int), cudaMemcpyHostToDevice
        );
    cudaMemcpy(d_v_strides, &v_strides, 2 *
         sizeof(int),
        cudaMemcpyHostToDevice);

    int threadsPerBlock = BASE_THREAD_NUM;
    dim3 blockDims(threadsPerBlock,
        threadsPerBlock, 1); // Adjust
        these values based on your specific
         requirements
    dim3 gridDims((max_lora_group_size +
        threadsPerBlock - 1) /
        threadsPerBlock, (n +
        threadsPerBlock - 1) /
        threadsPerBlock, n_lora);
    SGMVKernel<<<gridDims, blockDims>>>(
        d_v, d_v_shape, d_v_strides, d_in,
            d_in_shape, d_in_strides, d_a,
            d_a_shape, d_a_strides,
            d_lora_idx_s
    );
    cudaDeviceSynchronize();
    dim3 gridDims2((max_lora_group_size +
        threadsPerBlock - 1) /
        threadsPerBlock, (p +
        threadsPerBlock - 1) /
        threadsPerBlock, n_lora);

    SGMVKernel<<<gridDims2, blockDims>>>(
        d_out, d_out_shape, d_out_strides,
            d_v, d_v_shape, d_v_strides,
            d_b, d_b_shape, d_b_strides,
            d_lora_idx_s
    );

    // Copy back to the host
    cudaMemcpy(out, d_out, batch * m * p *
        sizeof(float),
        cudaMemcpyDeviceToHost);

    cudaDeviceSynchronize();

    // Check CUDA execution
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
      fprintf(stderr, "Matmul Error: %s\n",
          cudaGetErrorString(err));
      exit(EXIT_FAILURE);
    }

    // Free memory on device
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);
    cudaFree(d_out_shape);
    cudaFree(d_out_strides);
    cudaFree(d_a_shape);
    cudaFree(d_a_strides);
    cudaFree(d_b_shape);
    cudaFree(d_b_strides);
    cudaFree(d_in);
    cudaFree(d_v);
    cudaFree(d_in_shape);
    cudaFree(d_in_strides);
    cudaFree(d_lora_idx_s);
    cudaFree(d_v_shape);
    cudaFree(d_v_strides);
}
```

Listing 2. launch SGMV kernel in C

4 EXPERIMENTS

In this section, we describe how to integrate the Punica framework into minitorch and use the custom SGMV CUDA kernels that we implemented. Then we describe different experiment settings that were run for quantifying the performance improvement, and overhead. We investigate the result as the system scales in ablation studies.

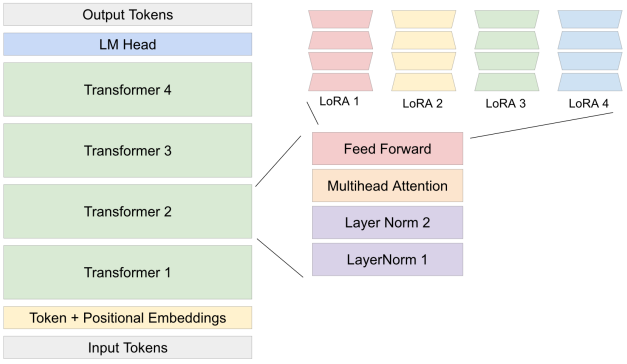

Figure 4. Customized DecoderLM with SGMV option enabled

4.1 Experiment Setup

We reuse the implementation of DecoderLM in Assignment 2 and connect our SGMV cuda kernels to minitorch by defining an additional SGMV Linear module with LoRA registration. We modified the Transfomer layer in the DecoderLM to provide the option for using SGMV in the feedforward layer. For simplicity when simulating serving

multiple LoRA, when the use_SGMV flag is set to be True, we randomly initialize different LoRA weights and set a fixed LoRA index registration.

To evaluate the performance improvement of adopting the Punica framework, we use the following experimental setup and try to analyze the performance improvement given the batching effect, memory optimization, and overhead. In each of the settings below, we generate 1000 distinct requests (40k tokens in total). The experiments are run on a single T4 GPU with 15.0 GB GPU RAM.

4.2 Batching Effect

To study the LoRA batching effect, we conduct experiments by varying the batch size and the number or LoRA that we are serving. On a single GPU, we served 2 or 4 LoRA with batch_size of 1, 16, and 32. In this setting, the 1000 distinct requests are uniformly distributed for every LoRA as we are stimulating a scenario where we have a lot of requests coming in at once and all uniformly distributed, this setting would allow us to analyze the best performance achievable by the batching effect. We record the total time required to process all the requests and present the results in Table 1.

n_lora	batch_size	Total Time (s)
2	1	8240
2	16	1895
2	32	1720
4	1	8375
4	16	1910
4	32	1720

Table 1. Batching Effect Result

4.3 Memory Optimization

To quantify the memory optimization by adopting the Punica framework, we calculated the number of parameters when serving separate LoRA each with their own underlying model as a baseline, and compared with using Punica to serve multiple LoRA with only one underlying model. We compute the numbers for serving 1, 2, and 4 LoRA to see how the memory optimization scales, the numbers are presented in Table 2.

Method	n_lora	of parameter (M)
Baseline	1	6.44
Baseline	2	12.89 (+6.45)
Baseline	4	25.78 (+12.89)
Ours	1	6.44
Ours	2	6.70 (+0.26)
Ours	4	7.23 (+0.53)

Table 2. Total Number of Parameters with different n˙lora

4.4 Overhead

To evaluate the overhead, we investigate the worst case where the 1000 distinct requests come in 1 request at a time and all the requests are for a single LoRA. This setting would allow us to quantify the overhead of additional logic added to support batch processing. We record the total time required to process all the requests when serving 1, 2, and 4 LoRA to see how the overhead scales. The results are listed in Table 3.

n_lora	Total Time (s)
1	8240
2	8265
4	8375

Table 3. Total time required to process all requests one by one

5 Analysis

5.1 Less time required

From the experiment, we observed that the average time for serving 1000 distinct requests to generate the full sentence is significantly reduced by enabling batch processing. The improvement from batch_size 1 to batch_size 16 demonstrated a 4x times 4x-performance improvement from better GPU utilization. The performance improvement for further increasing the batch_size to 32 however, diminishes, we suspect this is because as batch_size increases, the performance is bottlenecked by the GPU processing power instead of the GPU utilization.

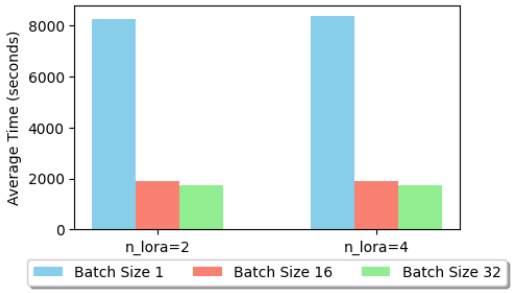

Figure 5. Average time for 1000 requests by batch˙size and n˙lora

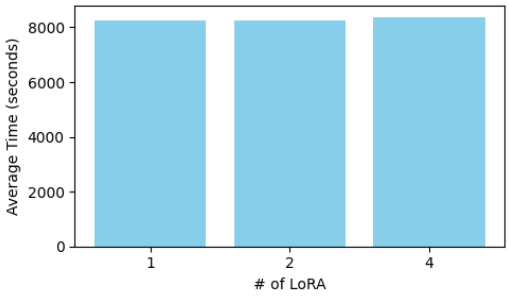

Figure 7. Average time for 1000 requests for single LoRA with batch size 1

5.2 Fewer of parameters

From our calculation of the total number of parameters when serving multiple LoRA separately compared with serving through the Punica framework, we see a significant reduction in the number of parameters, up to 4x fewer parameters for 4 LoRA, and the memory improvements continuous to scale as the number of LoRA increases, this is because the baseline method would require a new copy of the underlying model and the new LoRA weight, whereas with Punica framework, only the new LoRA weight would be needed given that the GPU can hold the additional compute resource requirements.

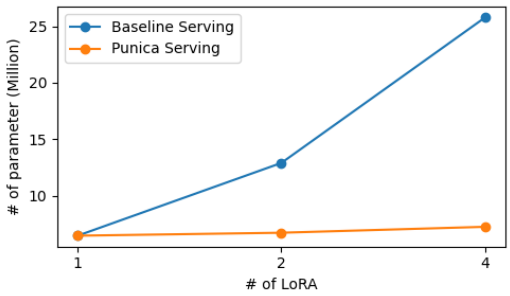

Figure 6. Number of parameters when serving multiple LoRA

5.3 Little overhead

From the experiment, we observed very little overhead when serving one request at a time to a single LoRA even when the number of LoRA served increases. This suggests that the Punica framework, under the worst-case scenario, can still provide comparable performance when the system scales to serve more LoRA.

6 Conclusion

Low-Rank Adaptation (LoRA) is an effective and efficient method to adapt pre-trained models to different domains, it is natural to explore the idea of serving multiple LoRA at once. However, there is little exploration in serving multi-tenant LoRA with only one GPU, and the common method is simply serving multiple copies of the underlying model with the LoRA weight, and processing requests one by one. These methods result in low GPU utilization and high memory requirements, we adopted the Punica framework and implemented the custom SGMV CUDA kernels, which enable the batch processing of requests to multiple LoRA. We integrate the CUDA kernels with minitorch and experiment on modified DecoderLM, the results show up to 4x less time and 4x less memory requirements for serving 4 LoRA with little overhead. In our current implementation, we still need to copy LoRAs from CPU to GPU for every run. For future work, we can explore using pybind+needle to keep LoRAs in GPU.

7 Team Member Contributions

Zengliang Zhu implemented the SGMV CUDA kernal and Launch_SGMV function in C, and corresponding sections in reports and the poster.

Xinran Wan contributed to writing the motivation sections for poster and other sections in the report, along with some preliminary debugging of the starter code.

Kath Choi contributed to the debugging of CUDA kernels, and integration of the CUDA kernels with minitorch and conducting all experiments and ablation analysis, in addition to writing the experiment and analysis section across all reports and the presentation poster.

REFERENCES

Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., and Krishnamurthy, A. Punica: Multi-tenant lora serving, 2023.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

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

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

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

Lester, B., Al-Rfou, R., and Constant, N. The power of scale for parameter-efficient prompt tuning. In Moens, M.-F., Huang, X., Specia, L., and Yih, S. W.-t. (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.243. URL https://aclanthology.org/2021.emnlp-main.243.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

Wang, Y., Ma, D., and Cai, D. With greater text comes greater necessity: Inference-time training helps long text generation. *arXiv preprint arXiv:2401.11504*, 2024.

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

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