ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH zürich

Semester Project

Systems Group, Department of Computer Science, ETH Zurich

End-to-End Optimization for FineInfer

by
Michael Kürsteiner

Supervised by
Prof. Dr. Gustavo Alonso, Yongjun He

June 2024–September 2024

D INFK

Abstract

Large Language Models (LLMs) keep increasing their capabilities and so does their demand. There is a need for sophisticated serving systems to allow a large number of users to utilize these models in their everyday lives. Inference requests are rather expensive compared to standard keyword queries and companies seek to reduce their costs by utilizing their GPU infrastructure to its fullest potential. One of the key metrics they optimize for is throughput. There are many approaches to achieve this, such as Continuous Batching[21], PagedAttention[13], FlashAttention[9], general kernel level optimizations and many more. In this work we develop our own LLM serving system including a command-line-interface (CLI), a server and an engine using Continuous Batching and we describe the design of every component in detail. We integrate an existing memory management approach inspired by PagedAttention, that reduces memory waste on GPUs and compare its benefits to our basic design. This design lies the foundation for our main goal of addingt support for fine-tuning tasks in the future for such a system. The addition of fine-tuning tasks improves utilisation of idle GPU resources. Otherwise unused GPU memory can now be used to process fine-tuning tasks. This introduces a new set of challenges requiring dynamic memory management.

Contents

1 Introduction **2**

2 Background **3**
- 2.1 LLMs and Transformers . . . 3
- 2.2 Continuous Batching . . . 4
- 2.3 PagedAttention and TokenAttention . . . 4
- 2.4 Fine-Tuning and LLM Serving Systems . . . 5

3 Design **6**
- 3.1 CLI . . . 6
- 3.2 Server . . . 7
- 3.3 Asynchrnous LLM Engine . . . 8
  - 3.3.1 Running Batch . . . 8
  - 3.3.2 Transformers KV Cache . . . 9
  - 3.3.3 Request Manager . . . 9
  - 3.3.4 Recv Loop . . . 10
  - 3.3.5 Engine Loop . . . 10
- 3.4 KV Cache Manager (Nopad) . . . 10
  - 3.4.1 Memory Manager . . . 12
  - 3.4.2 TokenAttention Request Manager . . . 13
  - 3.4.3 Triton Kernels and Batch Preparation . . . 14
- 3.5 Challenges and Known Issues . . . 17

4 Evaluation **19**
- 4.1 Setup . . . 20
- 4.2 Results . . . 21

5 Future Work: Dynamic Memory Management for FineInfer **25**

6 Conclusion **26**

1 Introduction

Large Language Models (LLM) are improved continually. Parameter sizes keep increasing under resource constrained environments. Companies seek to maximize their performance and minimize their cost. Hardware accelerators such as GPUs are critical for these systems and take up a large portion of the available resources. A single LLM request can be up to 10x more expensive than a simple keyword query[17]. As a result, optimizing the throughput of such LLM serving systems is of utmost importance to many companies.

When a user submits a request, the LLM generates the answer word (token) by word one at a time. Every new generation of a token is dependent on the input (prompt) and the previously generated tokens before that. This sequential process is repeated until the model generates a termination token. Each iteration adds new state that has to be kept in memory as the generation of the next token depends on this state, which is one of the main reasons why this problem is *memory bound*. The memory consumption of this state is mainly determined by the number of attention layers and their dimensions of the autoregressive Transformer model[19] in use, but also by the length of the sequence that is being processed. A sequence encompasses all tokens of the prompt and all newly generated tokens after that.

Available GPU resources are highly underutilized when processing single requests in that manner. However, batching of requests can significantly improve this bottleneck and up the throughput of the system. One of the most important factors that determines throughput is based on the maximum batch size[13]. When the dynamic state of every request is managed properly, the batch size can be pushed to its limits and increase throughput significantly. The dynamic state, usually referred to as *KV cache*[16], contains the key and value tensors that represent the context from previously generated tokens. Many LLM serving systems statically allocate GPU memory based on a predetermined maximally allowed sequence length[21, 14], hereby wasting up to about 80% of available memory[13] and negatively affecting their serviceable batch size and throughput.

Current deeplearning frameworks require tensors to be stored contiguously[15], but the KV cache is very dynamic in nature, as a priori, the lengths of the prompt and the final results are not known and can vary highly from request to request. As a result, memory waste is caused by internal and external fragmentation due to always preallocating the maximum sequence length contigously in physical memory for every request. The request might not use all the space for the generation of its final answer and newly scheduled smaller requests cannot make use of these remaining unused chunks but also need to preallocate a large chunk of memory wasting even more resources and further reducing maximum batch size and throughput.

LLM serving systems also observe idle times or have spare memory even if the system is running inference. These spare resources can be used to run finetuning tasks to improve both resource utilization and future quality of generated answers. In FineInfer [10] such an approach is outlined with their *Deferred Continuous Batching* algorithm. The combination of *Deferred Continuous Batching*

with a KV cache manager needs careful handling to securely provide resource efficient inference and fine-tuning, where the priority especially lies on fast serving times for inference tasks.

For this project, we explored current LLM serving systems[13, 2] to address many of the issues described before and built our own serving system, heavily inspired by current techniques. This includes a command line interface (CLI), a server and an engine driving the generation process of answers. Incoming requests are scheduled using Continuous Batching without any modified GPU kernels, entailing the need for padding of incoming requests to accommodate for the tensor shape of the currently processed batch. We measure the overhead of our design and integration of the CLI, server and engine and identify its main sources. We further integrate LightLLM's KV cache manager[2] into our own serving system and conduct an experiment to compare the normalised latencies and average batch sizes of our system with and without the KV cache manager. The last section of this report gives an outline of the challenges that lie ahead for the implementation of a dynamic memory manager that can combine LightLLM's KV cache manager with fine-tuning.

2 Background

The following section presents the current state of LLM serving systems and the techniques they apply to solve many of the issues as described in 1. The key mechanism we look into include the architecture of Transformer models[19], continuous batching[21], PagedAttention[13], TokenAttention[2] and FineInfer[10].

2.1 LLMs and Transformers

In essence, the main goal of a language model is to approximate the probability of a sequence of tokens $(x_1, x_2, ..., x_n)$. Due to the sequential nature of natural language, this is typically done by factorising the joint probabilities over the whole sequence as the product of their conditional probabilities (*autoregressive decomposition*[8]):

$$P(x) = P(x_1) \cdot P(x_2|x_1) \cdot ... \cdot P(x_n|x_1, ..., x_{n-1}). \tag{1}$$

Equation 1 directly implies the critical dependency that leads to the iterative process generating token after token in a sequential manner. Transformer models calculate these probabilities with the help of so called *self-attention* layers, which first apply a linear transformation to an input sequence $(x_1, x_2, ..., x_n) \in \mathbb{R}^{n \times d}$ at each position i to obtain the query, key and value vectors:

$$q_i = W_q x_i,\ k_i = W_k x_i,\ v_i = W_v x_i. \tag{2}$$

Followed by the computation of attention scores a_{ij}, as well as the outputs o_i as weighted averages over the value vectors:

$$a_{ij} = \frac{exp(q_i^\top k_j/\sqrt{d})}{\sum_{t=1}^{i} q_i^\top k_t/\sqrt{d}},\ o_i = \sum_{j=1}^{i} a_{ij} v_j. \tag{3}$$

Having a rough overview of the computations performed helps to understand the specialised GPU kernels needed to implement a proper KV cache manager. For other operations applied in embedding, normalization and feed forward layers as well as output logit computations, we refer to [19].

The generation process of an output sequence can be decomposed into two stages:

1. **Prefill stage**. In this stage, the key vectors $k_1, ..., k_n$, value vectors $v_1, ..., v_n$ and next output token x_{n+1} for an input prompt $(x_1, ..., x_n)$ are generated and saved. As the input embeddings for the input prompt vectors $x_1, ..., x_n$ are all known beforehand, the computation from equation 2 can make use of the inherent parallelism in tensor operations that are executed on a GPU.

2. **Decode stage**. After the prefill stage, the generation of the next token can be sped up by reusing the previously cached key and value vectors $k_1, ..., k_n$ and $v_1, ..., v_n$. This process is repeated until a termination token, also called end-of-sequence token, is generated. Due to the dependent nature of the iterative decoding stage, no parallelisation can be applied. In addition, for a single request, it only executes matrix vector multiplications and thus makes subpar use of available GPU resources.

2.2 Continuous Batching

In order to redeem the underutilisation of GPUs in the decoding stage, input prompts from multiple requests can be batched together to improve the overall token throughput. We also want to minimize the time spent for any request sitting in an idle queue. For that reason, serving systems like Orca[21] introduced an iteration level scheduling of newly arriving input prompts, such that they can be batched together with the current batch that is being processed. Instead of having to wait for a whole batch to finish, single terminated sequences can be removed from this batch to make space for new input prompts. Iteration-level scheduling is another way to improve the throughput of the system. Usually, incoming requests would need to be padded to the current shape of the batch to be able to concatenate them. Note that Orca makes use of custom GPU kernels to avoid this padding, preventing extra waste of GPU memory and thereby increasing possible batch sizes.

2.3 PagedAttention and TokenAttention

Continuous batching is able to boost throughput of LLM serving systems substantially, but it disregards a lot of the potential for improvement by not properly managing its KV cache. Even when applying batching techniques, the serv-

ing system is still *memory-bound*, greatly affected by the necessity of caching key and value vectors.

PagedAttention[13] applies the key concept of virtual memory known to every modern operating system[12]. This allows them to store logically contiguous key and value vectors non-contiguously in physical memory. In their notation, a page corresponds to a block, which contains the key and value vectors for a predetermined number of tokens. A block table maintains a mapping from a token of a sequence to the block and the offset within this block with the corresponding key and value vectors of that token. With this strategy, memory waste is constrained to one block, enabling larger batch sizes and higher throughput.

LightLLM[2] takes this a step further with their TokenAttention. Instead of allocating blocks, they allocate space for key and value vectors on a token level, eliminating even more memory fragmentation and thereby, allowing for even larger batch sizes.

Additionally, both vLLM (PagedAttention) and LightLLM (TokenAttention) developed specialised GPU kernels to enable computations with values stored in non-contiguous physical memory. vLLM has written their kernels using the C-like programming language CUDA, whereas LightLLM uses Triton[3]. The main advantage of Triton is its ease of usability and integration into Python. In the end, Triton code gets translated into CUDA code nevertheless. Using CUDA directly allows for more fine-grained control over memory allocation and reduce fragmentation further. They also apply other ideas such as parallel sampling, beam search, prefix sharing, preemption[13], FlashAttention[9] and others to further increase the throughput and quality of generated answers.

2.4 Fine-Tuning and LLM Serving Systems

Parameter-efficient-fine-tuning (PEFT) techniques[4] enable serving systems to save time and resources by reducing the number of trainable parameters by a large factor. The core LLM shares a base model and subsequent fine-tuned inference and training is executed by making use of multiple low rank LoRA adapters[11] at the same time. When switching tasks, instead of having to replace the whole model, the base model can be multiplexed and only a small subset of parameters for fine tuning have to be replaced. This minimizes the overhead penalty of switching between inference and fine-tuning tasks. This benefit opens up the opportunity to introduce real time fine-tuning tasks into existing LLM serving systems, that can use this idea to increase GPU utilization.

The two extreme approaches to support fine-tuning in a LLM serving system consist of either always prioritising inference requests or letting currently running fine-tuning tasks finish before the system continues with inference. The first method would constantly interrupt fine-tuning tasks whereas the second would violate service level agreements for guaranteed low inference latencies. FineInfer's *Deferred Continuous Batching* solution sits right in the middle. Instead of having to wait for the whole fine-tuning task, or at least a couple of fine-tuning epochs to finish, switching of tasks can be applied on an iteration level. They define a deferral bound with which they decide whether or not to

add a new inference request to the inference phase. If request's time spent on the waiting queue plus a time estimate of one fine-tuning iteration is larger than the deferral bound, the request will be added to the batch that currently is being processed for inference. Otherwise it stays in the queue and if the batch is empty, the system schedule an iteration of fine-tuning.

3 Design

Up next we discuss the design of the single components that make up the end product of this semester project. We start with the CLI, the server and the engine. For these components, the focus lies on making sure that requests can be scheduled using Continuous Batching and overheads introduced by the different components to the LLM serving system are within reasonable bounds. After, we take a look at LightLLM's KV cache manager design[2], how we integrated it into our own serving system and close it up with a description of challenges we have encountered and any known issues that need to be addressed in the future. We provide a flow chart in Fig. 1 to give a better overview of the components we developed and how they are connected.

3.1 CLI

The easiest way to interact with the LLM serving system for typical users is via the CLI. We designed this interface with two key ideas in mind: First, the user needs to be able to submit whole batches of input prompts that can be processed in the background while he is chatting with the LLM itself. Secondly, finished sequences should be displayed as fast as possible and while the user is chatting with the LLM, generating answers for the current conversation should be prioritised and not interrupted by answers of whole batches submitted before or during the chatting. It would be confusing to the user to get back a whole batch of answers while waiting for the answer to a single chat request. Our solution introduces a *batching* and a *chatting mode*.

When in batching mode, the user can submit batches of prompts to the server without waiting for the finished sequences. We accomplish this by using the *aiohttp* library that allows us to make non-blocking asynchronous requests to the server. The server immediately responds with a unique id for the submitted batch. The CLI is connected to a server socket, waiting for incoming finished sequences from the server in the background and it is also responsible of demultiplexing incoming responses to their according batches and keeping track of the finished sequences of a batch. As soon as a batch is finished, it prints the result on the terminal grouped by their unique batch id, given the user is in batch mode and otherwise keeps it stored until entering batch mode.

In chatting mode, the user can only submit one blocking request at a time to the server. The CLI will wait for the answer of the chat request and then display it to the user. While the user is waiting, he cannot submit any other requests, but all batch requests up until this point will be processed and the

chat request will be scheduled with a high priority together with the current running batch by the LLM engine to ensure low waiting times.

Requests in batching mode have to be submitted in the following format: ["Example Prompt 1", "Prompt 2", ...] [32, 64, ...]. The user includes all prompts in square brackets separated by commas and each prompt is enclosed in quotation marks. In the second pair of square brackets, the user can specify the maximum number of tokens that should be generated per request as integers. This can also be omitted and will default to a predefined value if not provided. The CLI notifies the user of any incorrectly formatted inputs.

3.2 Server

We developed the server using *FastAPI*[5] in order to keep it as simple as possible. In its current design, the server only handles one user at a time and is responsible for the following three tasks:

- Initialisation of services in separate processes. In the beginning, the server starts the engine process that prepares the LLM with all its parameters and sets up the sockets required for communication between the server and the engine. The server connects to that socket and creates its own socket for message passing with the CLI. Using multiple processes for different components allows for simple adding of new components to the system. Additionally, by communicating over sockets instead of inter process communication paradigms, a lot of the message passing is simplified.

- Listening to incoming requests from the CLI of the user, generating unique batch ids and responding to the CLI with these ids immediately, as well as forwarding the request to the engine.

- Listen for any finished sequences from the engine and transmit them to the CLI.

Following the design of LightLLM, for our sockets we use pyzmq[6], a Python library providing bindings for ZeroMQ, a high-performance asynchronous messaging library. It enables scalable, distributed, and concurrent communication between processes or across networked devices using various messaging patterns like push-pull or pair communication. Combining pyzmq with FastAPI has the advantage to provide a responsive and simple server API for processing incoming and outgoing data and fast message passing between processes to minimize communication overhead. Without the zmq sockets, the user would need to either constantly poll for any finished sequences, which scales poorly with a large user count in the future, or use a potentially slower socket API that would also complicate the communication between other components of the LLM serving system.

3.3 Asynchrnous LLM Engine

Following the main goals of our initially mentioned design principles, the architecture of the LLM engine supports scheduling of batches on an iteration level, processing of batches in a first-come-first-serve manner, prioritising chat requests over batching requests, queuing of incoming batches from the server, the automatic dispatch of waiting batches to the model as soon as possible while avoiding to run out of memory on the GPU, concatenating batches dispatched from the waiting queue with the current running batch and removing finished sequences from the running batch to finally send them to the server. All this is done in a concurrent manner with the help of the Python's *asyncio* library. In principle, a general metadata object for the current batch (BatchMeta object), a dictionary called **model_kvargs** with metadata to maintain the KV cache, two loops, and a request manager for scheduling are required in order to run all of the steps of the engine.

3.3.1 Running Batch

The engine stores its intermediate outputs and associated metadata in a dictionary called **running_batch**. Within a running_batch the actual token_ids of every sequence are stored in the **"input_ids"** field and it also contains a **BatchMeta** object with the following fields:

- **prompt_lens**. A torch tensor containing the number of tokens of every input prompt in a batch.
- **gen_lens**. A torch tensor containing the maximum number of new tokens to be generated of every prompt in a batch.
- **cur_lens**. A torch tensor containing the current number of total tokens for every sequence in a batch ignoring any padding.
- **ids**. A tensor where every element is a tuple of the form (b_id, r_id) denoting the unique batch id and request id of every sequence in a batch.
- **unfinished_sequences**. A boolean tensor acting as a mask to filter out the finished sequences.
- **stopping_criteria**. An object to determine when a batch of sequences is finished. This can be modified to include other criteria to determine when a generation process for a sequence should be stopped, e.g. based on output scores or different eos-tokens.
- **timestamps** A float64 tensor that stores the arrival times of every sequence in a batch as well as time spent of every sequence in actual inference. This allows to measure total latency and and overhead of the engine and other components per sequence.

Using PyTorch tensors enables effortless arithmetic, concatenation and masking operations to update the running batch during every engine step.

3.3.2 Transformers KV Cache

The **model_kvargs** dictionary is needed in order to operate the LLM as served by the Transformers library. It contains the intermediate KV cache with its key and value vectors and an additional *attention mask* required by its attention layers. In the later design 3.4, the model_kvargs dictionary becomes obsolete and will be replaced with a set of different structures.

3.3.3 Request Manager

The request manager is a critical component of the LLM serving system and defines the strategy to schedule new requests. It is also responsible for updating and maintaining the running_batch, KV cache and priority waiting queue. Our LLM engine was designed to require a request manager that must provide at least the following functionality:

- **add_reqs()**. Takes two batches and their associated meta data and concatenates them into one batch ready for inference.
- **remove_reqs()**. Removes any finished sequences from the running_batch and any associated meta data with it, frees up memory on the GPU and sends the output to the server.
- **get_fitting_batches()**. This method employs a strategy to determine how many new batches can be scheduled for inference by predicting the current memory requirements of the running_batch and and the new batch to add. As such, it takes the running_batch as an argument for its prediction.
- **waiting**. A priority queue for arriving batches.

For our first design, we built the **SimpleReqManager**. In order the concatenate two batches, the add_reqs method determines which batch is the one containing the longest sequence. All sequences in the batch with the smaller maximum sequence lengths are padded with pad_tokens to that length. Now both batches have the same shapes and can be concatenated. Additionally, the same principle is applied to all its metadata including the KV cache stored in the model_kvargs. This approach is responsible for a lot of memory waste in the system.

In the get_fitting_batches method, batches are simply popped from the priority waiting queue until we hit the memory limit. Batches that fit into memory are concatenated and dispatched to the prefill stage and the last batch that did not fit is put back on the priority queue based on its batch id. Estimations for future memory requirements by the running_batch and a potential new batch always assume that every sequence will potentially generate the maximum number of allowed tokens. A new batch will only be added to the running_batch if there is enough space to support the generation of an answer with maximum output length, as we do not know these output lenghts in advance. Again, this

wastes a lot of resources by not scheduling batches that could easily fit into memory.

On initialisation, the SimpleReqManager needs to be configured with a maximum number of tokens that can be held in GPU memory to serve the KV cache and additional metadata as described before. This number describes the memory limit of the system in terms of maximum number of allowed tokens in the running_batch.

3.3.4 Recv Loop

Its only task is to listen for any incoming batches from the server on the corresponding zmq socket using a pull-only pattern and call the **put** method of the engine. The put method prepares the **BatchMeta** object for the batch and puts it into a priority waiting queue based on its batch id. The batch id serves as a unique identifier and gives a stream of incoming batches an order in which they are supposed to be processed in a first-come-first-serve fashion.

3.3.5 Engine Loop

At its core, the engine loop can roughly be described with the pseudo code in 1. When no requests are being processed, the engine loop simply waits using an **asyncio event**. Whenever a new batch is enqueued via the **put** method in the **recv_loop**, the waiting event gets notified and the engine loop continues execution. After every completed engine step, the loop gives up execution using the **asyncio.sleep(0)** method for a brief moment. Without it, the loop would run forever and prevent the engine from queuing up new incoming batches.

During an engine step, the **add_reqs** and **remove_reqs** methods provided by the **req_manager** are used to update the running_batch and any associated metadata. The **generation_step** method differentiates between a batch in *prefill* and in *decode* stage in order to generate the KV cache for inputs in the prefill stage and reuses the cache for token generation during the *decode* stage. For that matter, it takes a second argument that is true if the batch to be processed is in prefill stage and else false for the decode stage. Of critical importance is the **get_fitting_batches** method. It decides which batches can be added to the running_batch and ensures we don't run into out-of-memory (OOM) issues during the engine loop.

3.4 KV Cache Manager (Nopad)

Proper management of the KV cache enables the serving system to make better use of its GPU memory. LightLLM's implementation stores the logical mappings of sequences in a contiguous table that maps each token_id to its corresponding key and value vectors in physical memory (KV cache). Instead of having to pad any incoming batches to the size of the current length of the longest sequence in the running_batch and waste a large amount of memory by allocating space in the KV cache for many unused tokens, we can can now make better use of

Algorithm 1 Async Engine Loop

```
procedure ENGINE LOOP
    reqs_in_progress ← False
    running_batch ← empty_batch()
    while True do
        if req_in_progress == False then
            wait_for_new_reqs()

        reqs_in_progress ← engine_step()
        await sleep(0)

procedure ENGINE STEP
    new_reqs ← req_manager.get_fitting_batches(running_batch)
    if len(new_reqs["input_ids"]) ≥ 1 then
        new_reqs ← generation_step(new_reqs, True)
        running_batch ← req_manager.add_reqs(running_batch, new_reqs)

    if len(running_batch["input_ids"]) ≥ 1 then
        running_batch ← generation_step(running_batch, False)

    if has_finished_sequences() == True then
        running_batch ← req_manager.remove_reqs(running_batch)

    reqs_in_progress ← len(running_batch["input_ids"]) ≥ 1
    return reqs_in_progress
```

our available GPU memory. We follow up with a discussion of its additional requirements and detailed implementation.

As the key and value vectors are stored non-contiguously in physical memory, we cannot simply make use of the LLM as provided by the Huggingface[1] library, which is a platform providing a large variety of different machine learning models via a comprehensive API called Transformers[7]. Most models in use were constructed using PyTorch, which does not provide fine-grained control over its tensor operations to allow for seamless integration of such a mechanism. For this reason, existing LLM serving systems[2, 13] developed their own kernel to support inference with non-contiguous key value vectors.

LightLLM built up the models they offer from scratch and developed a kernel for every operation performed in the layers during inference with the Triton library. They manually setup the model with the corresponding layers and parameters themselves, reusing the weights and configurations obtained from Huggingface. We used their code and integrated it into our own serving system. However, the more interesting part consists of the additional requirements to make the mapping from logical token_ids to the KV cache in physical memory work.

The integration of the KV cache manager into our own serving system required some reverse engineering. LightLLM additionally employs a more complex cache, namely a radix cache for advanced optimisation techniques. We did not work with the radix cache but focused on the base implementation of the simpler KV cache manager component. To picture the mapping of logical to physical memory, we provide Fig. 2 and the following sections explain all the additional parts and modifications to the existing design that are needed for this basic implementation.

3.4.1 Memory Manager

On initialisation, the memory manager is configured with a **size**, a **head_num**, a **head_dim** and a **layer_num**. The *size* describes the maximum number of tokens that can be served by the KV cache and the other variables denote the dimensions of the LLM layers to deduce the shape of the KV cache. Here are the important structures and functionality provided by the memory manager:

- **mem_state**. Initialised as int32 tensor with *size* zeros. It serves as a reference counter to every location of the KV cache.
- **indexes**. A int64 tensor with indices ranging from 0 to $size-1$. Facilitates the allocation of contiguous memory.
- **kv_buffer**. The non-contiguous KV cache as a list of tensors. Each tensor in the list saves the key value vectors for its corresponding layer.
- **alloc()/alloc_contiguous()**. These methods return the indices to the allocated spots in physical memory for the key and value vectors. The allocated spots are registered by incrementing reference counters to the *mem_state* tensor at the corresponding indices.

- **free()**. Takes a tensor of indices indexing the tokens in physical memory that can be removed and decreases the reference counters at the corresponding spots in the *mem_state* tensor.

3.4.2 TokenAttention Request Manager

Contrary to the memory manager 3.4.1, the request manager needed to be modified in order to satisfy the constraints as imposed in 3.3.3. Its configurations requires a *max_request_num* and a *max_sequence_length* to initialise its state. This section introduces the key components of the TokenAttention request manager:

- **req_state**. A bool tensor of size *max_request_num* to keep track of the maximum number of requests that can be processed at the same time in GPU memory.
- **req_to_token_indexs**. An int32 tensor of shape [max_request_num, max_sequence_length]. In essence, this is the table that holds the indices that map a token from logical memory directly to the corresponding index in the *kv_buffer*. The maximum number of requests and sequence lenghts should be chosen with the dimension of the *kv_buffer* in mind. For further use, we call this tensor the **cache indices table**.
- **mem_manager**. The memory manager as described in 3.4.1.
- **waiting**. A priority queue for arriving batches.
- **free()/alloc()**. Methods to reserve and free space on the *req_state* tensor and *kv_buffer*. The *alloc()* method returns a tensor containing the indices pointing to the cache indices table.
- **Modifications**. The BatchMeta object is updated to include a tensor (*req_cache_idxs*) with the indices for to the cache indices table as returned by the *alloc()* method. The *add_req* and *remove_req* no longer need to take care of the *model_kvargs* dictionary. The *get_fitting_batches* method is adapted to predict the dispatch of batches according to the available space on the *kv_buffer* tensor.

In Fig. 2 an example of a mapping from logical tokens to their key/value values in physical memory is depicted. For this example the maximum allowed sequence length equals 4 and a maximum of 4 requests can be processed at the same time. Each request of the *running_batch* is assigned a *req_cache_idx* that maps to the entry in the *req_to_token_idxs* table. Every row of this table corresponds to one request with all its token indices pointing to the key/value vectors in physical memory (*kv_buffer*). The order of the mapping process is given by the roman numbers I-VI. The upper part of Fig.2 shows an example state of this mapping process and the lower part depicts the updated state after one generation step. The request with *req_cache_idx* 1 is finished and removed from the *running_batch* together with its entries in the cache indices table and

its key/value vectors in the KV cache. For the other two sequences a new spot (in red) for each is allocated on the KV cache and the cache indices table to process the next token in each sequence.

3.4.3 Triton Kernels and Batch Preparation

LightLLM developed a series of Triton kernels in order to support inference on unpadded input prompts and a non-contiguous KV cache. For future work, it is important to understand how such a kernel is built and what extra information is needed to correctly extract the key/value vectors token by token. Before every generation step, the batch to be processed needs to be prepared with additional metadata for the kernels to work properly. In the following, we present a small Triton example, showing on a basic level how a kernel works and then continue to explain how a batch is prepared for inference.

Example Kernel: RMSNorm. The code snippet below is taken from the Triton kernel for the RMSNorm layer of the Llama LLM written by the LightLLM developers. One of the simpler kernels was chosen to give an easier to understand introduction into the API of Triton.

```
@triton.jit
def _rms_norm_fwd_fused(
    X,  # pointer to the input
    Y,  # pointer to the output
    W,  # pointer to the weights
    stride,  # how much to increase the pointer when moving
        by 1 row
    N,  # number of columns in X
    eps,  # epsilon to avoid division by zero
    BLOCK_SIZE: tl.constexpr,
):
    # Map the program id to the row of X and Y it should
        compute.
    row = tl.program_id(0)
    Y += row * stride
    X += row * stride
    # Compute variance
    _var = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        x = tl.load(X + cols, mask=cols < N,
            other=0.).to(tl.float32)
        _var += x * x
    var = tl.sum(_var, axis=0) / N
    rstd = 1 / tl.sqrt(var + eps)
    # Normalize and apply linear transformation
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        mask = cols < N
```

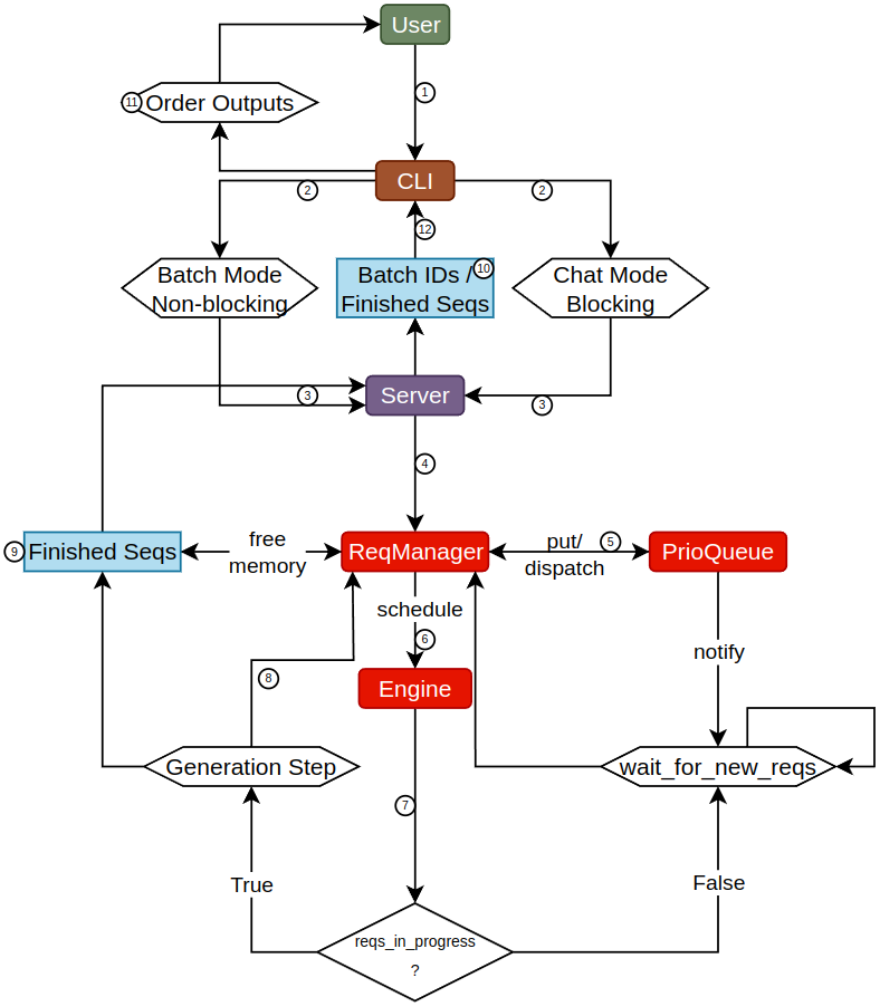

Figure 1: Flowchart giving an overview of the LLM serving system. The numbers in small circles show an example flow of a user input through the serving system.

```
27          w = tl.load(W + cols, mask=mask).to(tl.float32)
28          x = tl.load(X + cols, mask=mask,
                other=0.).to(tl.float32)
29          x_hat = x * rstd
30          y = x_hat * w
31          # Write output
32          tl.store(Y + cols, y.to(Y.dtype.element_ty),
                mask=mask)
```

This kernel uses one worker for every row of X, if there are M rows then we have M workers. First, it finds the offset into X for the current worker by extracting the id of the worker in line 12 as *row*. By multiplying it with the *stride* we adjust the pointer from the beginning of X to the row that the worker is assigned to (line 14). The same is done for the Y pointer. After that, every worker proceeds in a block-wise manner. The size of each block is defined as *BLOCK_SIZE*. The loop starting at line 17 loads *BLOCK_SIZE* numbers of X into x, making sure that any indices that are out of range result in a load of a zero (line 19) and then the necessary operations are executed to obtain the *rstd* value (line 22). The second loop starting at line 24 proceeds in the same manner, additionally storing the results to the location pointed at by the adjusted Y pointer (line 32). The kernel can then be called following the example of this snippet below:

```
1  def rmsnorm_forward(x, weight, eps):
2      # allocate output
3      y = torch.empty_like(x)
4      # reshape input data into 2D tensor
5      x_arg = x.view(-1, x.shape[-1])
6      M, N = x_arg.shape
7      BLOCK_SIZE = 128 * 2 * 2 * 2 * 2 * 2 * 2 * 2
8      # enqueue kernel
9      _rms_norm_fwd_fused[(M,)](x_arg, y, weight,
10                               x_arg.stride(0), N, eps,
11                               BLOCK_SIZE=BLOCK_SIZE)
12     return y
```

Note that x needs to be reshaped into a matrix, as it consists of a 1-dimensional tensor containing the input token of every sequence concatenated together without any padding. When calling the kernel, one has to provide a grid that describes the arrangement of workers expected for the kernel. This is done in line 9 by adding $[(M,)]$ after _rms_norm_fwd_fused. Hereby, we tell the kernel we expect a set of workers arranged in a 1-dimensional grid of size M, such that we can map every worker to one row of *x_arg*, as described in the first snippet.

Batch Preparation. Every Triton kernel of the attention layer needs to be provided with additional information. LightLLM stores this in a class object called **InferState** and it includes the following metadata:

- **b_req_idx**. A tensor containing the indices for every request in the batch pointing to their corresponding entries in the cache indices table.

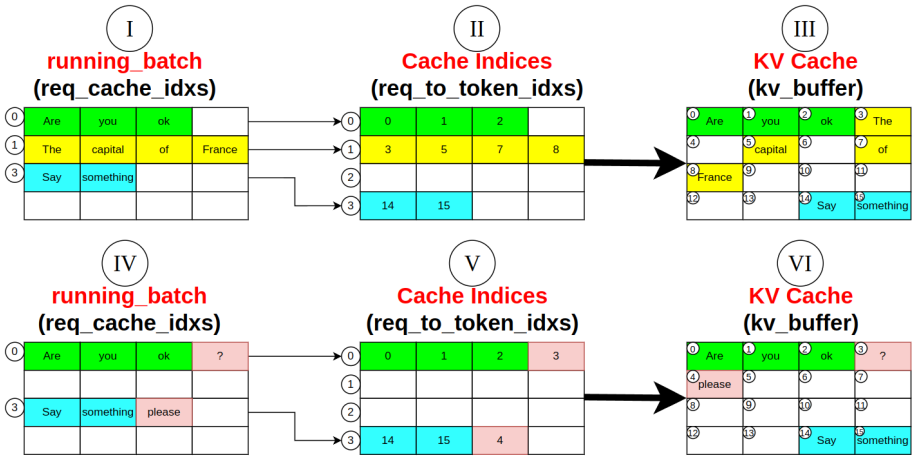

Figure 2: Example of showing how the mapping from logical tokens stored contiguously in the running_batch to its corresponding key and value values in physical memory.

- **b_start_loc**. A tensor denoting the start locations of the different sequences within the batch.
- **b_seq_len**. A tensor containing all current sequence lengths in the batch.
- **is_prefill**. A single bool to determine whether a batch is in its prefill or decode stage.

There are dedicated methods to generate this additional information for prefill and decode batches right before the batch is sent to the model for inference. By including this information, we can use our serving system with unpadded inputs, give the kernels all the additional metadata they need to correctly calculate the offset of each sequence within a batch and find the correct key/value values in the non-contiguous KV cache.

3.5 Challenges and Known Issues

We outline challenges and known issues that need to be addressed in future work. It is not an exhaustive list but it reflects our expected behaviour for the serving system.

Scheduling granularity. Currently, the request manager is not capable of scheduling single requests of a submitted batch. Allowing for single requests in a batch to be scheduled would allow us to push the average running_batch size even further in some cases. There are multiple possible locations to tackle this issue. For example, we could decompose every submitted batch into single

requests at the CLI level or modify the *get_fitting_batches* method of the request manager to look at every batch on a per request granularity. As such, there is also no logic that prevents a user to submit a batch so large, that it will never get scheduled because it simply does not fit into memory. The request manager will never dispatch it for that reason and the engine would stop, as batches are processed in FCFS manner and the batch that is too large would block other batches from being dispatched as well.

Automatic configuration according to available memory. Memory that is preallocated at startup does not make optimal use of its available resources. It rather uses rough approximations on maximum batch sizes and sequence lengths from tests that we ran until the system throws an out-of-memory (OOM) error. These estimations leave some performance on the table. To determine its exact memory limits, more measurements have to be done to squeeze out the last bit of available memory from the system. This is not as easy to measure as the CUDA kernel allocates additional memory for every tensor and also other intermediate tensors created during inference. The preallocation process of the *kv_buffer* and maximally supported token number and any other state for the KV Cache manager design do not take this fact into account. If not handled carefully, we could accidentally run out of memory during inference leading to the termination of the engine loop.

Quality and correctness of LLM answers. Determining whether a generated answer is correct or not can sometimes be tricky. Real world datasets can contain requests that are too long for our serving system to be processed in their full length and provided outputs are usually from a different model such as ChatGPT. This makes it hard to make comparisons between outputs and decide whether they are reasonable or not. For that reason, we tested our implementations by comparing them to the output of the unmodified HuggingFace model and its *generate* method. No pre- and post-processing or other optimization strategies, such as beam-search etc., are applied. We still encounter certain test cases where the generated results from our serving system do not match with the output of HuggingFace's generate method. The occurrence of such mismatches stays below 10% of the total number of sequences in our tests but varies depending on what requests are used within the test. In some cases, we still produce a reasonable answer but it is just different from the baseline. In other cases, the answer is unreasonable in both the generate method and our implementation by either not generating an answer or it keeps repeating the same few tokens over and over just with a different frequency between the two compared outputs. Which leads us to the next issue of missing pre-, post-processing steps and any other optimisations (parallel search, beam search, etc.) that can lead to an increase in quality of answers. Although this is not the focus of this project, it is still an important aspect of a serving system and should be considered.

BatchMeta and InferState. Our current design decouples the information contained in the BatchMeta class and the InferState class as introduced by LightLLM. This unnecessarily complicates the readability of the code. Combining these into one would remove redundant metadata and simplify the code. We also need to compare the overhead of the preparation of an InferState object versus the overhead of maintaining a BatchMeta object that handles updates with PyTorch tensor operations.

Server. There is a known issue, when a request contains a special character that leads to a premature termination of serialization process of a submitted request. This seems to be happening on the transmission path from the server to the engine via the according zmq socket. The result is the transmission of an incomplete batch of requests, obviously leading to different, wrong or missing answers. Lastly, the server should support multiple users at the same time.

Support for multiple LLMs. The design with the SimpleReqManager can easily handle various LLMs as provided by HuggingFace as long as they fit onto one GPU. However, the KV cache manager design does not. For the latter you need to load the parameters of the desired LLM and build up the layers yourself. The reason for this is, that the KV cache manager design requires special kernels and as each LLM is built differently, kernels need to be adapted to support all operations for a specific LLM and its layers. In our current design, the RequestManager is part of the engine. For the future it makes more sense to couple the request manager to the model. This makes it easier to support multiple models, as we can modify the RequestManager per model, instead of having to make modifications to the engine, which potentially could break other parts of the generation process for other models.

Exact overhead measurements. In section 4.2 we try to identify the main causes of the overhead for the engine and the KV cache manager. The profiling for the single functions in Table 4.2 is based on their cumulative execution times and these functions also mix up CPU and GPU operations. Distinguishing between the two to exactly identify the main cause of overhead, for example in the generation step method, requires more fine-grained profiling to get a better measurement of the actual time spent on non-inference tasks. This can be achieved with a line-profiler in the future.

4 Evaluation

We conduct three types of tests. The first series measures the overhead of the engine itself, the server and the CLI. We also compare the overhead of the engine with the SimpleReqManager (also referred to as "Engine Only") and the engine with the KV cache manager, which we call the Nopad design for the remainder of this section. We evaluate the performance of both designs on real world datasets to see the benefits of a KV cache manager in terms of throughput with

varying request rates. For the third category we measure the average batch size of the running_batch for both designs under full load.

4.1 Setup

Model and GPU. We use LLaMA[18] with 8B parameters. The GPU is a RTX 3090 with 24 GB of memory.

Metrics. The main focus lies on throughput. However, we report the *normalized latency* of the designs as the mean of the end-to-end latency of the request divided by its output length as done in Orca and PagedAttention [21, 13] for the second set of tests. This metric gives us better insight if the serving system is overwhelmed and many requests spend a lot of time in the waiting queue under increasing request rates. Throughput alone does not reflect that aspect, as it stays more or less the same throughout the tests and when under full load.

LightLLM implemented their TokenAttention kernels in combination with FlashAttention and we were not able to detach them from each other. For that reason the measurements show the benefit of both TokenAttention and FlashAttention. We also measure the average batch size at every generation step of the running_batch to give a better view on the benefits of the KV cache manager in isolation. The overhead evaluation measures the latency of a single batch of requests. The results are split into overhead latency and forwarding latency.

Workloads. The normalised latency and batch size measurements are run on the ShareGPT and Alpaca datasets, which contain input sequences of real LLM services. ShareGPT is a collection of user-shared conversations with ChatGPT. Alpaca is an instruction dataset generated by GPT-3.5 with self-instruct[20]. According to the PagdAttention paper [13], the ShareGPT dataset has 8.4× longer input prompts and 5.8× longer outputs on average than the Alpaca dataset, with higher variance. On average an input from ShareGPT has 161.31 tokens and an input from Alpaca has 19.31 tokens and the average output lengths are 337.99 and 58.45 tokens respectively. To simulate arrival times of requests, we use a Poisson distribution. The tests for the average batch size are run under full load for 15 minutes per test.

The overhead test uses a single request that reliably produces the maximum amount of allowed tokens in the system. We submit a batch with 3 copies of this request and report the median out of 50 attempts per tested component. We measure the overhead for the example request with the following two input and newly generated token number combinations: (1024, 1024) and (256, 128). The first is chosen by the maximally allowed input and output lengths, whereas the second one strikes a balance between the smaller average input and output length distributions of the aforementioned datasets.

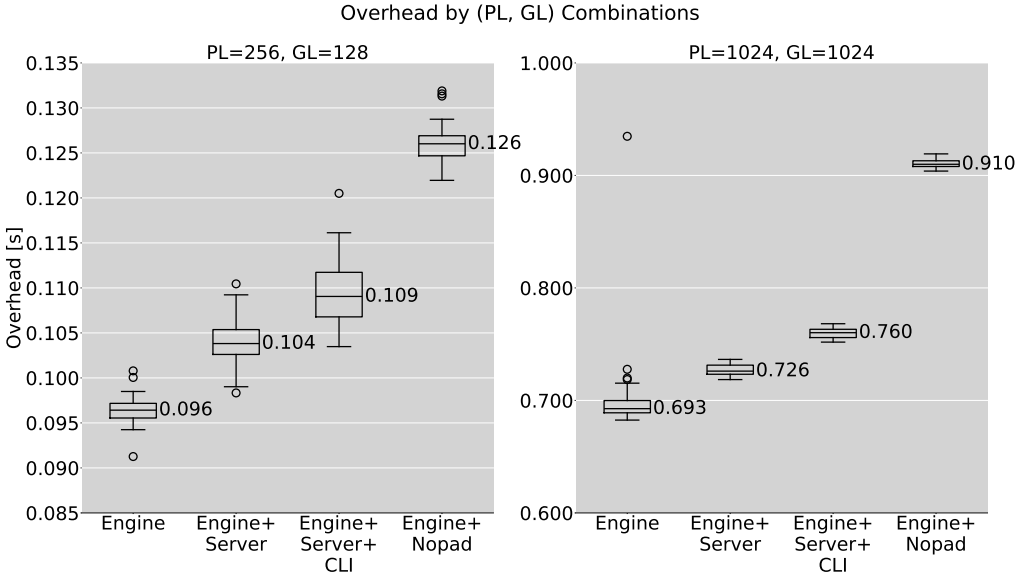

Figure 3: Overheads for two prompt and generation length combinations with medians out of 50 trials. Boxes represent the range from the first quartile to the third quartile. Outliers are 1.5× larger or smaller than that range.

4.2 Results

This section presents our data. Note that when we talk about the SimpleReqManager or about the engine on its own, we refer to the same design. This is different from the engine with the Nopad design, that has a KV cache manager.

Normalised Latency. The evaluations in Figures 5 and 6 show that the SimpleReqManager design reaches its limits in both datasets very quickly. Request rates of 0.24 reqs/s for ShareGPT and 0.3 reqs/s for Alpaca are already enough to increase the normalised latency severely. The Nopad design can sustain much higher request rates and only starts increasing normalised latency at around 2 reqs/s for ShareGPT and 4 reqs/s for Alpaca. This is explained by the larger capacity of the KV cache in the Nopad design, being able to accommodate up to 2.9× more tokens in total. FlashAttention further reduces the latency of every request, in turn also reducing the time a request spends in the waiting queue. To give a more fine-grained view, we also measured and compared the two designs for smaller request rates in Fig. 7 and Fig. 8.

Average batch size. The measurements in Fig. 9 show the benefit of efficient memory utilization by the Nopad design. Compared to the SimpleReqManager, the running_batch is about 4.7× and 4.5× bigger in the Nopad design

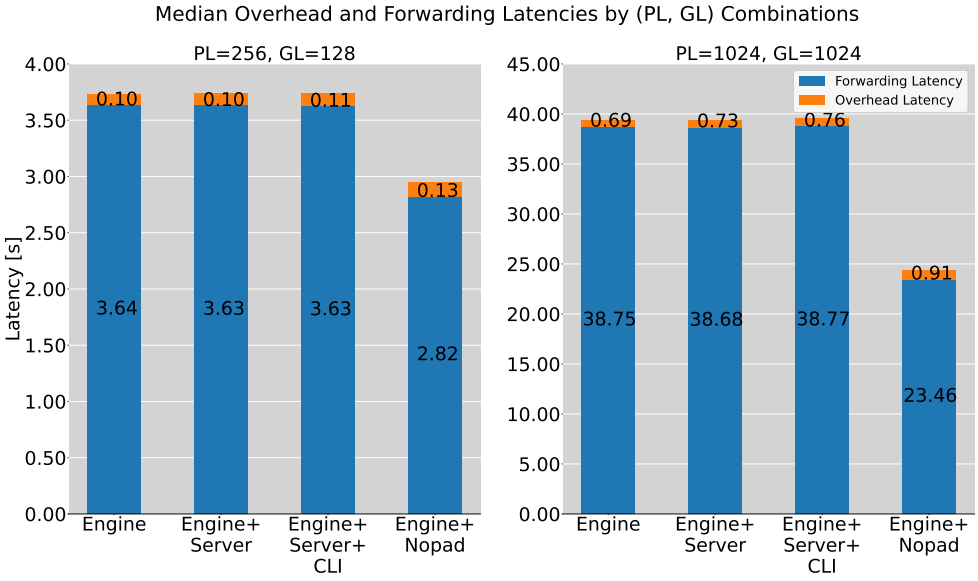

Figure 4: Median latencies seperated into forwarding and overhead latency.

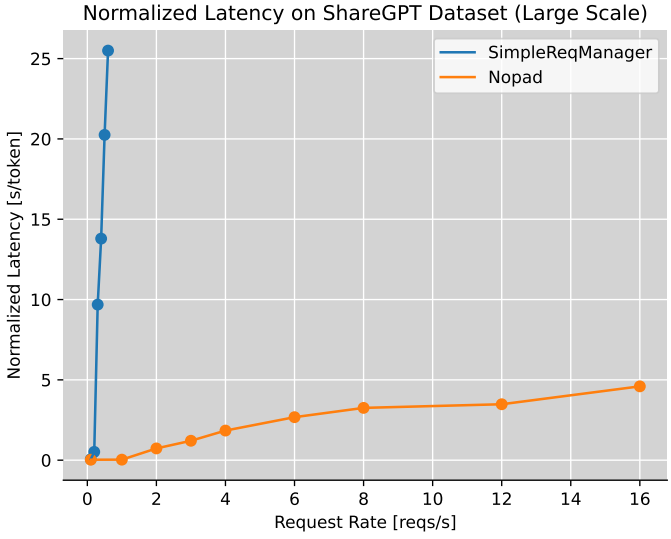

Figure 5: The capability of the two tested designs to maintain normalised latency under increasing request rates for the ShareGPT dataset.

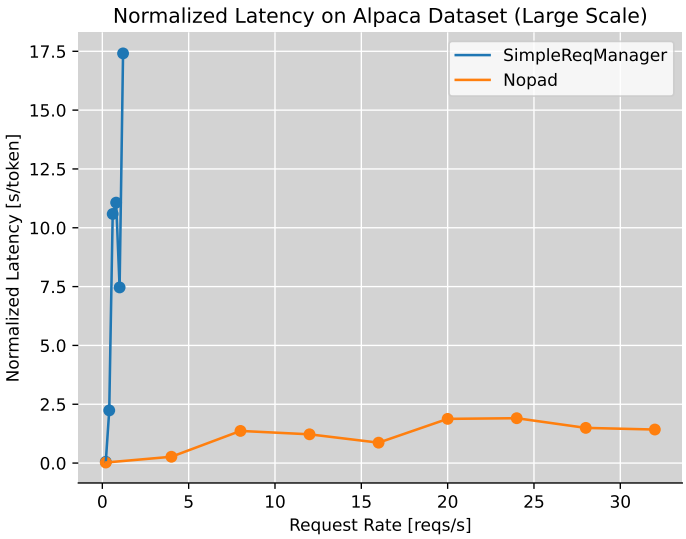

Figure 6: The capability of the two tested designs to maintain normalised latency under increasing request rates for the Alpaca dataset.

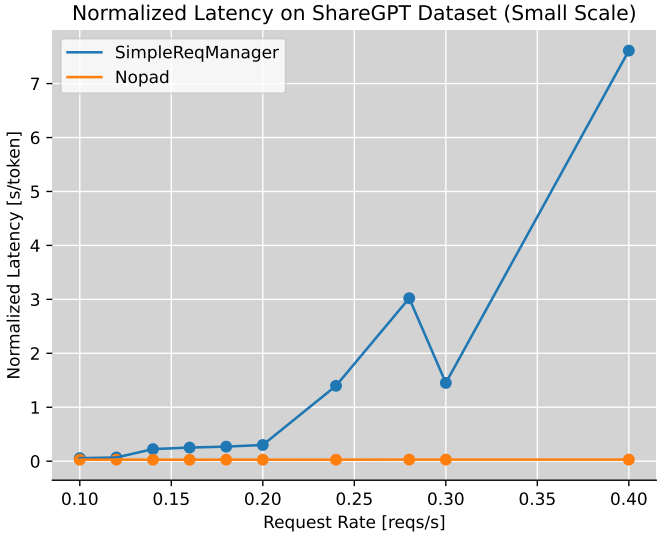

Figure 7: The capability of the two tested designs to maintain normalised latency under increasing request rates for the ShareGPT dataset.

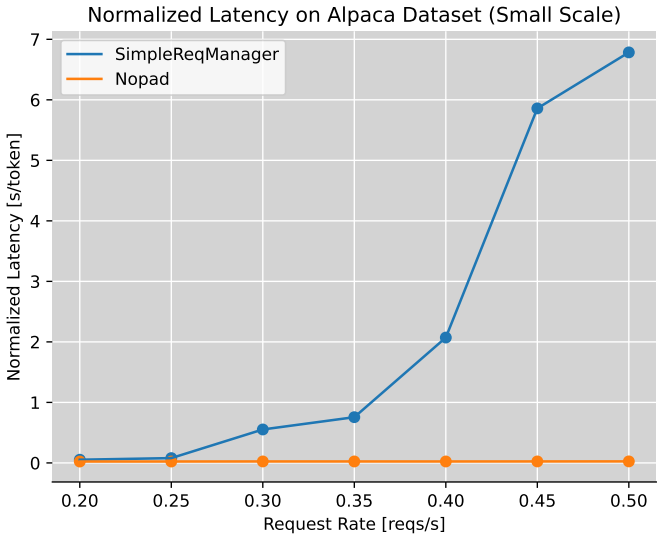

Figure 8: The capability of the two tested designs to maintain normalised latency under increasing request rates for the Alpaca dataset.

for ShareGPT and Alpaca datasets respectively. This result is purely based on the KV Cache manager itself and shows its benefits with no influence from FlashAttention. We account these numbers to the increased capacity of the KV cache from reduced allocation of memory during inference and the removal of memory fragmentation caused by padding of input sequences.

Overhead. The measurements in Fig. 3 and Fig. 4 confirm two things. First, the introduction of the engine, server and CLI increase latency only by a small fraction when compared to the forwarding latency. For smaller sequences we pay a slightly higher price relative to total latency as expected in terms of overhead. Second, the Nopad design has a 1.3× higher overhead for small and large sequences compared to the SimpleReqManager. In the Nopad design the allocation of physical memory, the maintenance of the KV cache with its indices to physical memory as well as preparing the *InferState* for the running_batch during each generation_step call are the main causes for this increase in overhead. In the "Engine Only" design, the main causes are the updates to the KV cache and model_kwargs and preparation of the inputs during every generation_step call step executed by the Transformers API. Checking for finished sequences using the stopping_criteria takes up a substantial portion of the total overhead in both designs during the generation_step call. We provide table 4.2 that lists the highest cumulative execution times that are the main cause of overhead for the "Engine Only" and the "Engine + Nopad" design. These times are from a

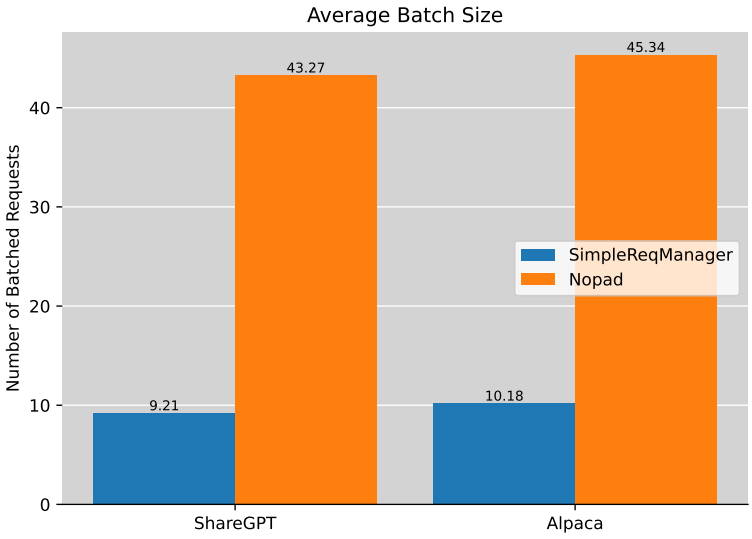

Figure 9: The average batch sizes of the running_batch for the Nopad and SimpleReqManager design on the ShareGPT and Alpaca datasets.

separate profiling run and are not the measurements shown in Fig. 3 and Fig. 4. We note that the tokenization of inputs takes place in the put method of the engine. This could be done outside of the engine, which would move a part of the overhead to a different component. More fine-grained measurements need to be done in order to identify the exact composition of the generation_step overhead.

5 Future Work: Dynamic Memory Management for FineInfer

The work described up until this point is supposed to serve as the basis to implement the combination of inference and fine-tuning tasks and utilize available GPU resources to their full potential. On a high level, the procedure we aim to implement follows the concept of FineInfer[10]. For future work, this concept would need to be implemented into the current design. This entails the support for PEFT techniques and scheduling fine-tuning tasks via the engine using deferral bounds. One of the key challenges is how to handle memory when we need to switch during a fine-tuning task back to inference and the inference requests need the memory we reserved for fine-tuning. We need a a way to recover the state for the fine-tuning task in order to continue when fine-tuning is scheduled again. This requires a dynamic memory management system that

Engine Only

Function	[s]
engine_step	**+0.682**
generation_step	0.658
other	0.612
forward (non-GPU)	0.046
other	0.015
remove_reqs	0.005
get_new_reqs	0.003
update_cur_pad_len	0.001
put (tokenization)	+0.006
engine_loop	+0.004
other	+0.096
Total	**=0.788**

Engine + Nopad

Function	[s]
engine_step	**+1.035**
generation_step	0.979
other	0.519
forward (non-GPU)	0.460
other	0.032
get_new_reqs	0.013
remove_reqs	0.007
put (tokenization)	+0.013
engine_loop	+0.004
other	+0.009
Total	**=1.061**

Table 1: Tables showing the cumulative overhead latencies of functions for the "Engine Only" and "Engine + Nopad" designs and a (1024, 1024) prompt/generation length combination. By overhead, we mean time spent on CPU-tasks to get an estimate of actual time spent on non-inference related overhead. Entries called "other" include lines of miscellaneous code such as torch.builtin methods, if statements and so on.

is able to manage memory for the KV cache and its mappings, adapters and additional state required for training, such as gradients. As starting point, a dynamic KV cache manager is required that is able to react to changes in the size of the available memory dynamically, regardless of the application that causes this change of available memory. After that, we can tailor its design to the fine-tuning scenario that takes into account the additional requirements for fine-tuning.

6 Conclusion

In this semester project we developed a LLM serving system. This system contains an engine running the main generation process of answers using Continuous Batching, a FastAPI server that forwards any requests to the engine and responds with finished sequences and a CLI through which a user can submit batched inference requests and chat with the LLM. All components work together asynchronously and were tested and evaluated in terms of their overhead and throughput. We provide detailed description of the components and outline their challenges and known issues. We then integrated LightLLM's KV cache manager into our serving system in order to remove the need for padding and eliminate a large chunk of resource waste caused by memory fragmentation. We reverse engineered the process of its index-bookkeeping and give a detailed description of its procedure. Our measurements show the improvements obtained

by this KV cache manager in terms of sustainable request rates and average batch sizes on two real world datasets. Compared to the SimpleReqManager, We see an increase in average batch size of about 4.7× in the Nopad design for the ShareGPT dataset. This shows the immense benefit of a KV cache manager in an LLM serving system. Finally, we identify what challenges lie ahead for the future implementation of a dynamic memory manager that handles inference with a KV cache manager and fine-tuning tasks simultaneously.

References

[1] Huggingface.

[2] Lightllm: A light and fast inference service for llm. https://github.com/ModelTC/lightllm. [Online; accessed 23-August-2024].

[3] Triton. https://github.com/triton-lang/triton. [Online; accessed 24-August-2024].

[4] Parameter-efficient fine-tuning. https://github.com/huggingface/peft, 2023. [Online; accessed 24-August-2024].

[5] Fastapi. https://github.com/fastapi/fastapi, 2024. [Online; accessed 9-September-2024].

[6] Pyzmq: Python bindings for Ømq. https://github.com/zeromq/pyzmq, 2024. [Online; accessed 9-September-2024].

[7] Transformers: State-of-the-art machine learning for jax, pytorch and tensorflow. https://github.com/huggingface/transformers, 2024. [Online; accessed 11-September-2024].

[8] Y. Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. volume 3, pages 932–938, 01 2000.

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[10] Yongjun He, Yao Lu, and Gustavo Alonso. Deferred continuous batching in resource-efficient large language model serving. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 98–106, New York, NY, USA, 2024. Association for Computing Machinery.

[11] 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, 2021.

[12] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F H SUMNER. One-level storage system. *IEEE Transactions on Electronic Computers*, EC-11(2):223–235, apr 1962.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[14] Nvidia. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM/tree/release/0.5.0?tab=readme-ov-file, 2023. [Online; accessed 23-August-2024].

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[16] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.

[17] Reuters. Focus: For tech giants, ai like bing and bard poses billion-dollar search problem. `https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/`, 2023. [Online; accessed 22-August-2024].

[18] 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, 2023.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[20] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions, 2023.

[21] 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. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.