Computer Science

Capstone Report - Fall 2022

Optimizing the Serving System for Large Language Model Inference

Yao Xiao

Supervised by
Guyue Liu

Declaration

I declare that this senior capstone was composed entirely by myself with the guidance of my advisor, and that it has not been submitted, in whole or in part, to any other application for a degree. Except where it is acknowledged through reference or citation, the work presented in this capstone is entirely my own.

Preface

This paper aims to address the challenges of serving large language models (LLMs) in real-time. It may be helpful to scholars who conduct research LLM-related system design, providing insights into how to override the KV cache bottleneck in LLM inference and how to mitigate pipeline bubbles in the architecture.

Acknowledgements

I would like to express my sincere gratitude to Professor Guyue Liu for her invaluable guidance, unwavering support, and insightful discussions throughout the development of this project. I am also grateful to the members of the research group led by Professor Guyue Liu for exchanging and discussing ideas, which has enriched my understanding and contributed significantly to the overall quality of this work. I extend my thanks to Professor Promethee Spathis for his valuable feedback and suggestions on this project. Finally, I deeply appreciate everyone who is not aforementioned but has supported me in this project, including my friends and family.

Abstract

Large Language Models (LLMs) such as ChatGPT have been progressively revolutionizing natural language processing tasks. Since most LLM applications such as chatbots and question answering systems involve real-time interactions, high throughput and low latency are critical for LLM inference systems. However, existing LLM inference systems stuggle because the batch sizes are limited by the key-value cache (KV cache) memory, and bubbles exist in the inference pipeline due to the Attention mechanism, leading to inefficiency in inference. In this paper, I present FluidInfer to address these challenges. FluidInfer abandons the term *batch size* and instead arbitrarily concatenates or splits requests in non-Attention layers to achieve much higher throughput. It further packs requests dynamically in the Attention layer to reduce pipeline bubbles, while overriding the limitation of KV cache by swapping between GPU and host memory. Simulation results show that FluidInfer outperforms state-of-the-art LLM serving systems by 2x to 4x improvement in throughput, and achieves approximately 3x lower latency bound.

Keywords

Artificial Intelligence; Large Language Models; Scheduling; Inference Systems; Distributed Systems; System Performance

Contents

1 Introduction **6**

2 Related Work **9**

3 Background **11**

4 FluidInfer Design **13**

4.1 KV Cache Limitation 14

4.2 Pipeline Bubbles 16

5 Implementation **19**

6 Evaluation **19**

6.1 Experimental Setup 19

6.2 Normalized Latency 21

6.3 Latency Bound 21

7 Conclusion **22**

1 Introduction

In recent years, LLMs have demonstrated exceptional capbabilities in a wide range of natural langauge processing tasks, such as chatbots [1, 2, 3] and coding assistants [4, 5]. However, along with their emergent abilities come increasingly massive model sizes [6, 7, 8], leading to extremely high computational and memory costs. For instance, GPT-175B requires 325GB of GPU memory simply to load its model weights even with parallelism and quantization techniques, let alone KV cache and activations during inference [9]. This would translate into at least 5 NVIDIA A100 GPUs, which cost over $\$100,000$ in total. Moreover, at the core of LLMs, the computational complexity of the transformer model is quadratic to sequence lengths [10]. Consequently processing a LLM request can be tens of times more expensive than a traditional query [11]. Given these high costs, it is thus critical to optimize the serving system for higher throughput and lower latency. However, there are two major challenges that existing systems struggle to address.

Firstly, KV cache is a bottleneck in LLM inference, limiting the maximum possible batch size thus limiting throughput. Figure 1 demonstrates the memory layout in the GPU when serving LLM inference, exemplified by using the OPT-6.7B model [3] with four NVIDIA GeForce RTX 4090 GPUs. The available space for KV cache is only 30% of the total GPU memory, while the KV cache size required by a single request with 2048 tokens of context length is nearly 8GB. This number would further increase linearly as the model size scales up. vLLM [12] has applied virtual memory management techniques such as paging and page swapping to KV cache to allow more efficient KV cache utilization, but it still cannot override the limitation of KV cache. In short, the batch size is limited by the KV cache size, and the throughput of existing systems is limited by the batch size.

To resolve this bottleneck, I propose to get rid of the notion of *batch size* by the following observations. First, the maximum possible batch size is determined by the KV cache, while Attention is the only operation in transformer models that require the presence of KV cache in GPU memory. Therefore, it is unnecessary to use the same upper bound for the number of requests that can be simultaneously processed in non-Attention layers and in the Attention layer. Secondly, the computations in non-Attention layers are token-wise, meaning that they do not need to distinguish the tokens from different requests. In other words, tokens from different requests can be arbitrarily concatenated, and those from the same request can also be arbitrarily splitted, as long as we keep track of how they are reassembled together. This approach would

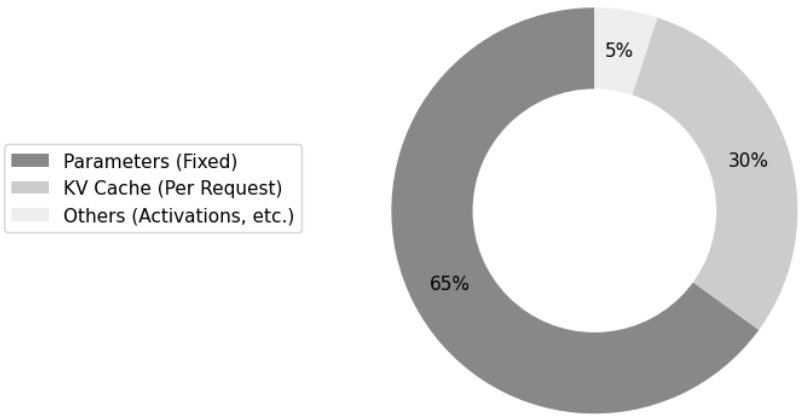

Figure 1: Memory layout when serving LLM inference, using the OPT-6.7B model [3] with four NVIDIA GeForce RTX 4090 GPUs as an example.

maintain exactly the same computation results, thus not sacrificing the accuracy of the model. As in continuous batching [13], the concatenated sequence will be splitted before Attention, passed through the Attention layer in parallel, and finally reassembled again in the same order for further layers. No longer being limited by the KV cache, it becomes possible to perform batched processing of non-Attention layers at much larger scales to achieve higher throughput.

However, the above approach introduces a new problem: the number of requests that need to go through the Attention operation in a forward pass can be much larger than the maximum number of requests whose KV cache can be held in the GPU memory at the same time. To address this issue, I propose to offload and upload KV cache between GPU and host memory. To mitigate the overhead incured by such cache swapping, a proactive swapping pipeline can be implemented to overlap memory communication with computation. This approach maintains the original level of parallelization in the Attention layer, thus not affecting its efficiency. Combined with the previous technique that accelerates non-Attention layers, this approach can achieve overall higher efficiency.

Secondly, state-of-the-art LLM inferences implement continuous batching [13], meaning that scheduling is done at the granularity of *iterations*, i.e., generating one next token. However, existing LLM inference systems do not distinguish iterations in different inference phases, leading to bubbles in the execution pipeline. In the *prefill* iteration when a request has just arrived, none of its tokens have been cached and we need to compute the keys and values for all its input tokens. On the other hand, in a *decode* iteration, all previous tokens of the request has already been cached so we need to compute the key and value only for one new token. Note that the computational

complexity of Attention is quadratic to the sequence length [10], so the computation time for these two phases varies significantly. Moreover, as a consequence of continuous batching, the Attention operation remains unbatched so different iterations are computed separately in parallel [13]. Hence, with prefilling iterations being significantly slower than decoding iterations, there exist pipeline bubbles that lead to inefficiency in inference.

To resolve this challenge, the system should be aware of iterations in different phases and approximates their Attention time by the number of input and cached tokens. In particular, a long prefill iteration can be executed in parallel with several sequentially scheduled short decode iterations. Also, if a long and a short prefill iterations are scheduled in parallel, the short decode iteration can further be sequentially packed with several decode iterations to minimize the pipeline bubbles.

Based on the aforementioned techniques, I present FluidInfer, a distributed serving system for the inference of transformer-based LLMs. FluidInfer is not fully implemented, but a comprehensive simulator is implemented to evaluate different methods and demonstrate its proficiency. The results show that FluidInfer significantly outperforms vLLM [12], showing 2x to 4x throughput improvement and approximately 3x reduction in latency bound. To summarize, the main contributions of this paper include:

- I propose to get rid of the notion of *batch size* by concatenating or splitting requests in non-Attention layers, overriding the limitation of KV cache to achieve higher throughput.
- I solve the issue due to overriding KV cache limit by offloading and uploading KV cache between GPU and host memory, and further implement a proactive swapping pipeline to mitigate cache swapping overhead.
- I identify pipeline bubbles in the Attention operation, and propose a novel scheduling algorithm to pack multiple iterations in the pipeline to minimize the bubbles.
- I simulate FluidInfer, the system that implements the aforementioned techniques, and demonstrate that is substancially outperforms the state-of-the-art solution vLLM [12].

Organization. In §2, I will present the related work regarding fast serving systems of LLM inference. In §3, I will briefly introduce some basics of LLM inference that are essential for further discussions. In §4, I will discuss in detail the design of FluidInfer, extending and explaining the aforementioned challenges and proposed solutions. In §5, I will briefly list the

implementation efforts of FluidInfer, though it is still under development. In §6, I will provide evaluations of FluidInfer to demonstrate its proficiency. Finally in §7, I will conclude this paper and discuss future works.

2 Related Work

Batch processing. One common approach to accelerate LLM inference serving is to batch multiple requests to increase throughput. Most LLM inference systems, such as TensorFlow Serving [14] and Triton Inference Server [15], implement this approach. This technique stacks the tensors of a batch of requests, leveraging hardware capabilities in parallel computing and reducing memory overhead by processing more data within a single load of parameters. However, naive batching results in the same batch size across all transformer layers, being suboptimal in that the limitation of batch size for the Attention layer is different from that for the non-Attention layers.

Memory optimization. KV cache is the bottleneck that bounds the maximum possible batch size, leading to limited throughput. Motivated by the bottleneck in KV cache utilization, vLLM [12] proposed an efficient KV cache management strategy, inspired from the paging technique in operating systems. It observed that current serving systems pre-allocate sufficient memory in the KV cache such that it can hold the maximum number of tokens of that request, because the Attention operation must be performed in contiguous memory. However, as is shown in Figure 2a, this causes serious memory overhead. Most requests never get close to the maximum generation length (light gray parts are significantly shorter than dark gray parts), leading to internal fragmentation. Moreover, external fragmentation may also arise, in that Request 3 cannot fit in even though there is theoretically sufficient memory space. vLLM suggests that the KV cache can be divided into KV blocks to hold keys and values just as dividing kernel space memory into pages to hold processes (Figure 2b). Internal fragmentations are resolved becacuse pre-allocation is no longer needed, and external fragmentations are minimized given reasonable block size. It also developed the PagedAttention algorithm to perform computation on each KV block and combine them together. Current LLM inference systems [16, 17] have widely adopted these optimization, as this paper will also follow. However, vLLM only addresses the KV cache utilization issue, but not the KV cache size limitation issue. This paper aims to override the limitation of KV cache and enable parallelized processing at much larger scales.

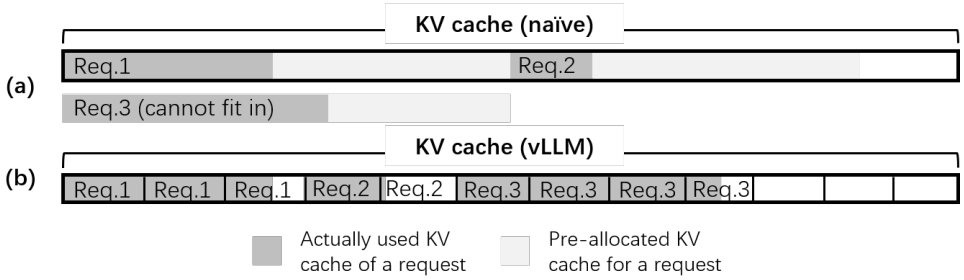

Figure 2: (a) The naive KV cache management, pre-allocating sufficient space for the maximum generation length (dark gray plus light gray). (b) The KV cache management of vLLM [12], dividing the KV cache into KV blocks.

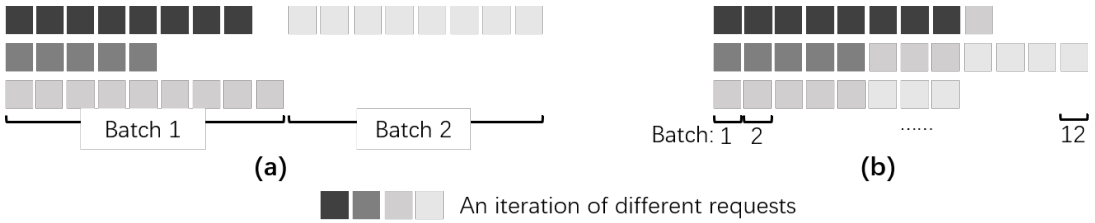

Figure 3: Each block represents an iteration of some requests, and different colors stands for different requests. The rows represent the requests in the a batch, and the widths of the blocks represent the execution time. Orca [13] assumes that each iteration takes approximately the same amount of time, hence same width. (a) Naive request-level batching and scheduling. (b) Continuous batching with iteration-level scheduling.

Scheduling. Reasonable and efficient scheduling is another critical component in serving LLM inference. Orca [13] pointed out that request-level batching can cause serious latency. Due to the autoregressive nature of LLM inference (i.e., generate one next token iteratively), inference time can largely vary due to different and unpredictable response lengths, even with the same model. Request-level batching can easily cause requests with short responses to be delayed if there are requests with long responses in the same batch (Figure 3a). Therefore, Orca proposed continuous batching, splitting a single query into iterations and thus allowing early-return of short queries and early-join of new requests (Figure 3b). The only issue with continuous batching is that the Attention layer cannot be batched due to varying tensor shapes, but experiments have shown that this have minimal effect compared with the benefits that continuous batching can bring. Recent systems [16, 18, 17] have gradually been supporting this feature for better performance, which is what this paper also adopts. On top of this, FastServe [18] improved the trivial first come first serve (FCFS) scheduling of Orca to avoid starving short requests due to no preemption. It suggests that multilevel feedback queue (MLFQ) can be utilized to approximate shortest remaining processing time first (SRPT) behavior in the information-agnostic setting where we do not know in advance the remaining time of a request.

However, none of these approaches distinguish iterations in different phases, leading to bubbles in the execution pipeline. Since a prefill iteration needs to compute the attention values of all input tokens, while each decode iteration needs to compute the attention value of only one new token, though the decode iterations take approximately the same time, there can be significantly gaps between decode and prefill iterations, or prefill iterations with different input lengths, which is the one of the problems that this paper aims to address. Sarathi [19] is the only current LLM serving system that is aware of these bubbles, but it only addresses this issue in the context of pipeline parallelism [20, 21] but not in general. Deepspeed-FastGen [17] released a blog showing awareness of this issue as well, but it has not released its technical details. This paper aims minimize these pipeline bubbles by designing a novel scheduling algorithm to pack multiple iterations in the pipeline, which will be a general solution under any setting.

3 Background

In this section, I will introduce the transformer architecture and the inference procedure of transformer-based LLMs, which will be the basis of the rest of this paper.

The transformer architecture. The transformer is the core of most LLMs nowadays [1, 2, 3]. Simply put, it does only one thing, i.e., generate the most-likely next token given a sequence of input tokens. Because of this, in order to generate a sequence of output tokens, the transformer model needs to be executed multiple times, each time generating one token, appending it to the input sequence, and repeating the process in the next iteration until a special end-of-sentence (`<eos>`) token is generated. As is shown in Figure 4, each column represents an iteration (or a *forward pass*) through the transformer model, consisting of N identical transformer layers, though with different sets of model parameters.

The structure of each transformer layer is further depicted in Figure 5. For simplicity, it demonstrates the single-headed non-batched scenario. Multi-headed Attention is essentially the same, and the effect of batching will be discussed later in this paper. Ignoring minor sublayers such as LayerNorm and AddResidual, each transformer layer consists of the following steps:

- *Pre-Attention:* A tensor of shape $[L, H]$ is fed into the transformer layer, where L is some sequence length and H is the hidden size of the model. It is passed through the QKVLinear sublayer, which is a matrix multiplication with a weight tensor $[W_Q \mid W_K \mid W_V]$ of shape

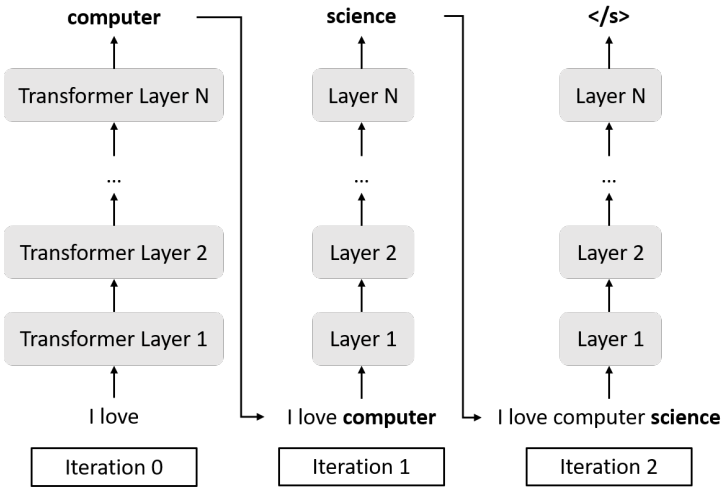

Figure 4: A computation graph of the inference procedure of transformer-based LLMs. For the sake of simplicity, it ignores non-transformer layers such as word embedding layer, etc.

$[H, 3H]$, and split into three tensors Q, K, and V, each of shape $[L, H]$, known as the query, key, and value.

- *Attention:* This is the operation that distinguishes transformer from other neural networks, allowing it to capture long-range dependencies. It computes the attention values as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{H}}\right)V, \tag{1}$$

namely the scaled dot-product attention [10].

- *Post-Attention:* The attention values of shape $[L, H]$ is passed through an OutProj sublayer, which is a matrix multiplication by the output projection weight W_O of shape $[H, H]$. After that, the feed forward network (FFN) module performs a matrix multiplication by W_1 of shape $[H, H']$, a non-linear activation such as ReLU, and another matrix multiplication by W_2 of shape $[H', H]$. Here, H' is the FFN dimension (also called the second hidden size) of the model. The output of the FFN module is again back to the shape $[L, H]$.

LLM inference procedure and KV cache explained. The inference of a transformer-based LLMs for each request consists of one *prefill* iteration (Iteration 0 in Figure 4) and multiple *decode* iterations (Iteration 1 and Iteration 2 in Figure 4). In a prefill iteration, L is the length of the input sequence. Particularly in the Attention sublayer, it follows the no-cache routine with $x = L$ in Figure 6 and computes the attention values of all input tokens, while storing their keys and values in the KV cache. The KV cache then plays its critical role in the decode iterations.

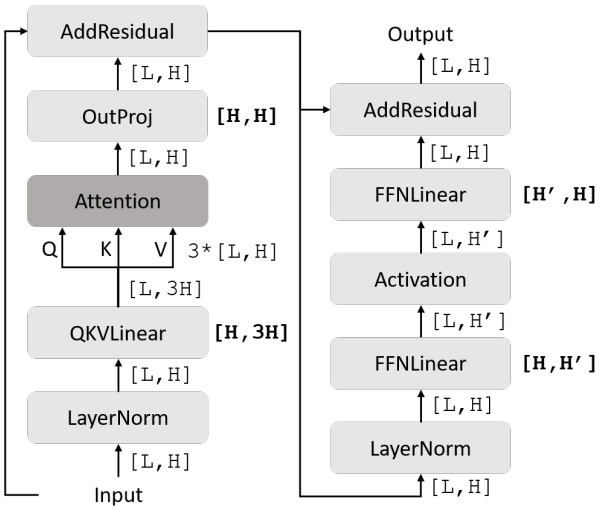

Figure 5: The structure of each transformer layer, which each node representing a sublayer. The shapes beside the arrows represent the shape of the input tensors to the next layer. If a sublayer is simply a matrix multiplication, the shape (bold) on its right represents the shape of the weight tensor.

In each decode iteration (suppose that input sequence has been appended to length x), only the attention value of the new token x is needed, so the data grayed out in the no-cache routine are redundant. However, the computation still requires the keys and values of all previous tokens, forcing an input tensor of shape $[x, H]$ instead of $[1, H]$. With the KV cache, however, keys and values of all previous tokens are cached to avoid recomputation, as is shown by the dark blocks in the cached routine in Figure 6. Consequently, an input tensor of shape $[1, H]$ would suffice in each decode iteration, trading space for efficiency and avoiding redundant recomputations. In particular, QKVLinear would output Q, K, and V all of shape $[1, H]$ in a decode iteration, but K and V are concatenated with the cache into shape $[x, H]$ before passing into the Attention layer. The output of the Attention layer remains of shape $[1, H]$.

4 FluidInfer Design

In this section, I will first introduce the mechanism to override KV cache limitation for higher throughput (§4.1). Then, I will present the scheduling algorithm to minimize pipeline bubbles in the Attention operation (§4.2). Finally, I will demonstrate how these techniques can be applied to a distributed serving system for LLM inference (§**??**).

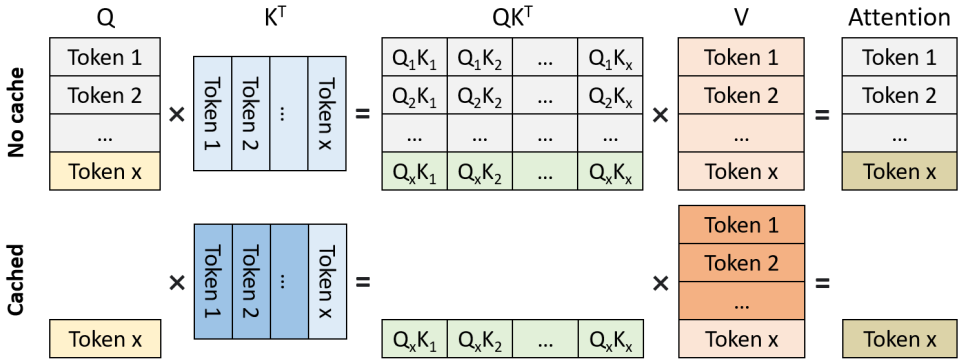

Figure 6: KV cache explained [22]. Operations in Attention other than matrix multiplications (e.g. Softmax) are not depicted for simplicity. The grayed out blocks in the no-cache routine (top) represent data that are needed in a prefill iteration but redundant in a decode iteration. The dark blocks in the cached routine (bottom) represent data that are cached and reused.

4.1 KV Cache Limitation

Challenge. In LLM inference, the available KV cache in the GPU memory is limited while each request can take a large amount of KV cache storage. Moreover, to execute a batch of iterations in parallel, it is required that the whole KV cache of the corresponding requests must exist in the GPU memory simultaneously. Consequently, the maximum possible batch size is limited by the KV cache size, and the throughput of existing systems is limited by the batch size. The ultimate goal is to override the limitation of KV cache, avoiding it from being the bottleneck of the system.

Observation. Firstly, the Attention layer is the only place where KV cache plays its role, while all other non-Attention layers does not require the corresponding KV cache to be existent in the GPU memory. Secondly, all the non-Attention layers are token-wise. In particular, Attention requires the notion of requests during matrix multiplications because it is not reasonable to compute the attention value between two tokens from different requests, while this is not the case for all other layers, including Linear, LayerNorm, AddResidual, Actication (e.g. ReLU), etc. Hence, it is possible to arbitrarily concatenate or split requests in non-Attention layers without affecting the correctness of the inference result.

Solution. Instead of executing the whole pipeline with a fixed batch size, FluidInfer gets rid of this notion and concatenate or split the input tokens of different requests arbitrarily. The execution pipeline of current LLM serving systems is shown as in Figure 7a. With a fixed batch size of 3 (which is the limitation of KV cache), 3 iterations are batched and executed in one forward

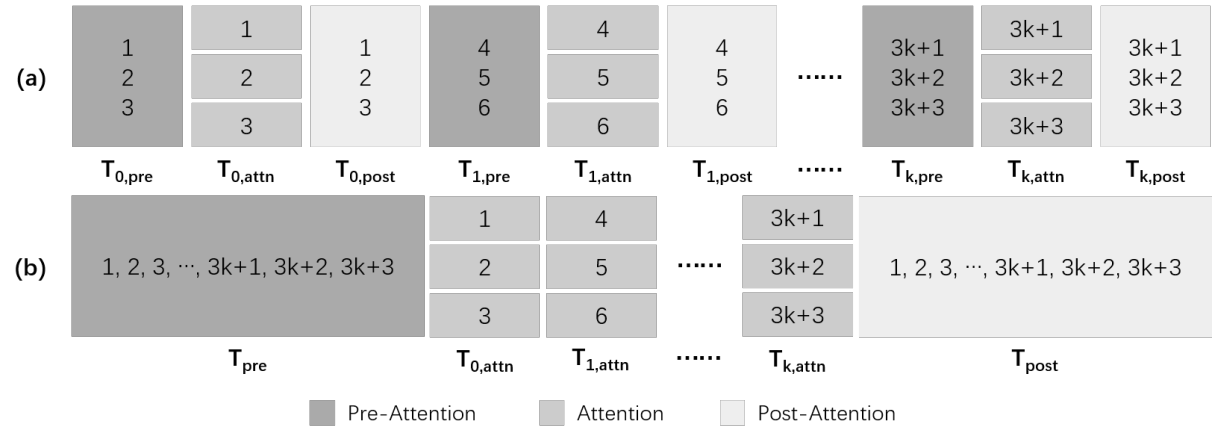

Figure 7: The execution pipeline, assuming that the GPU memory can hold the KV cache of at most 3 requests. The pipeline depicts only one transformer layer, since the case is the same for multiple layers. The numbers represent iterations from different requests, and different colors represent execution of pre-Attention, Attention, and post-Attention layers, respectively. A full block means concatenating input tokens of the iterations for computation. Separate blocks in a column means execution in parallel, with each block representing the execution of input tokens of one iteration. (a) The execution pipeline of current LLM serving systems. (b) The execution pipeline of FluidInfer.

pass through the model. Once the current batch of iterations is completed, the system schedules and executes the next batch of 3 requests. In Figure 7b, however, FluidInfer concatenates a much larger number of requests for pre-Attention and post-Attention layers. The exeuction of the Attention layer is the same as in existing systems, since it must be limited by the KV cache size. The reason why FluidInfer is more efficient is that, while Attention time remains unchanged, pre-Attention and post-Attention layers achieve higher level of parallelism. In particular, with the notations in Figure 7, the execution time of existing systems and FluidInfer are respectively

$$T_{\text{existing}} = \sum_{i=0}^{k}(T_{i,\text{pre}} + T_{i,\text{attn}} + T_{i,\text{post}}) = \sum_{i=0}^{k} T_{i,\text{pre}} + \sum_{i=0}^{k} T_{i,\text{attn}} + \sum_{i=0}^{k} T_{i,\text{post}}, \tag{2}$$

$$T_{\text{FluidInfer}} = T_{\text{pre}} + \sum_{i=0}^{k} T_{i,\text{attn}} + T_{\text{post}}, \tag{3}$$

where $T_{\text{pre}} \ll \sum_{i=0}^{k} T_{i,\text{pre}}$ and $T_{\text{post}} \ll \sum_{i=0}^{k} T_{i,\text{post}}$, especially with large k values. Optimally, the number of concatenated tokens should saturate GPU compute, requiring at least length of 512 under the setting of OPT-6.7B [3] with four NVIDIA GeForce RTX 4090 GPUs. Since each decode iteration consists of only one token, this is equivalent to a batch size of over 512, far beyond the maximum possible batch sizes supported by current serving systems, which would be at most 16 under the same setting.

KV cache swapping. In Figure 7b, the iterations $1, 2, \ldots, 3k+3$ are necessarily from different requests, because the next iteration in a request would depend on the result of the previous iteration in the same request, so they cannot be parallelized. Consequently, the number of iterations that need to pass through the Attention layer would largely exceed the KV cache limit, even though the number of iterations executed in parallel can be kept within that boundary. To overcome this issue, FluidInfer uses a KV cache swapping pipeline (Figure 8). During execution of the current iterations, FluidInfer offloads the KV cache of the previous iterations to the host memory, and uploads the next iterations to the GPU memory. For instance, when executing iterations 4, 5, and 6, the KV cache of iterations 1, 2, and 3 are being offloaded and that of iterations 7, 8, and 9 are being uploaded. Moreover, the memory communication overhead of KV cache swapping is minimized since offloading and uploading are overlapped with computation. A prefill iteration with input length L is compute-bound, which mainly involves a $[L, H] \times [H, L]$ and a $[L, L] \times [L, H]$ matrix multiplication, taking $4L^2H$ floating-point operations (FLOPs). This would take $\frac{4L^2H}{\mathrm{c}}$ nanoseconds, where c is the tera floating-point operations a GPU can compute per second (TFLOPS). A decode iteration with x cached tokens is memory-bound, which involves loading $4xH + 4x + 2H$ bytes from GPU high bandwidth memory (HBM) to GPU SRAM (Q, K, V, $QK^T/\sqrt{H}$, and softmax$(QK^T/\sqrt{H})$), and writing $4x + 2H$ bytes from GPU SRAM back to GPU HBM ($QK^T/\sqrt{H}$, softmax$(QK^T/\sqrt{H})$, and attention output). This would take $\frac{4xH+8x+4H}{\mathrm{BW}}$ nanoseconds, where BW is the bandwidth between GPU SRAM and HBM in GB/s. In comparison, the KV cache per token is $4H$ bytes, so swapping the KV cache of x tokens would take $\frac{4xH}{bw}$ nanoseconds, where bw is the bandwidth between GPU HBM and host memory in GB/s, commonly PCIe. For commonly used GPUs such as NVIDIA GeForce RTX 4090 and NVIDIA A100, the cache swapping time is approximately equal to or slightly larger than the execution time of a decode iteration, but much smaller than the execution time of a prefill iteration. The more iterations that can be executed in parallel in the Attention layer, the more likely a prefill iteration will exist, thus overlapping the KV cache swapping time and minimizing the memory communication overhead.

4.2 Pipeline Bubbles

Challenge. Continuous batching [13] eliminates pipeline bubbles caused by long requests blocking the return of short requests and the arrival of new requests. However, it does not distinguish iterations in different phases and with different sequence lengths, still leaving bubbles in the At-

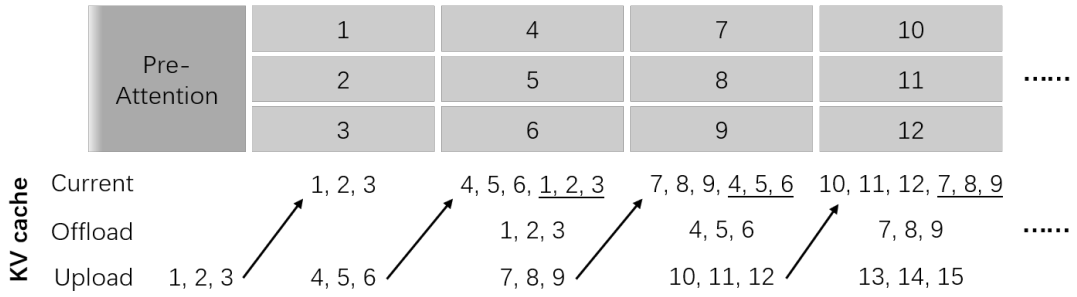

Figure 8: The KV cache swapping pipeline for Figure 7b. The numbers represent iterations from different requests. The underlined iterations in the current KV cache represent those whose KV cache are being evicted, and the rest are those whose KV cache are being used by the current computations.

tention computation pipeline. There are mainly two types of bubbles in current LLM serving systems (Figure 9a):

- *Type A:* Bubbles caused by varying prefill lengths. The computation of the Attention layer in prefill iterations have L equal to the input sequence lengths, mainly involving two matrix multiplications $[L, H] \times [H, L]$ and $[L, L] \times [L, H]$, thus taking $4L^2H$ FLOPs. Hence, different L values can have significant effect on the computation time in that the complexity is quadratic, leaving bubbles in the execution pipeline.

- *Type B:* Bubbles caused by prefill iterations executed in parallel with decode iterations. As is aforementioned, decode iterations has $L = 1$, thus performing $[1, H] \times [H, x]$ and $[1, x] \times [x, H]$ matrix multiplications, taking $4xH$ FLOPs where x is the number of cached tokens. Compared with $4L^2H$ FLOPs in prefill iterations, the complexity is linear, thus the computation time is much shorter, leading to pipeline bubbles. Note that different x values between decode iterations can also cause bubbles, but it is less significant than the aforementioned two types.

These pipeline bubbles are wasted computation resources and directly correspond to loss in serving system throughput. Hence, the ultimate goal is to mitigate these pipeline bubbles.

Observation. Firstly, different from pre-Attention and post-Attention layers where iterations of different requests are concatenated together for computation and the actual computation parallelization is up to CUDA, the Attention layer executes different iterations in parallel, meaning that a thread that completes its computation early can return immediately and start the computation of some other iteration, making it possible to pack multiple short iterations with a single

Figure 9: The execution pipeline, being aware of varying exeuction time in the Attention layer among different iterations, and ignoring pre-Attention and post-Attention layers for simplicity. (a) The execution pipeline of current LLM serving systems. Dotted boxes represent different types of pipeline bubbles. (b) The execution pipeline of FluidInfer.

long iteration to fill the bubbles. Secondly, since we override the KV cache limitation, there can be many more iterations available for packing instead of having only B iterations available, where B denotes the batch size. Hence, it is more likely to find a find iterations of different lengths to construct a bubble-free pipeline.

Solution. FluidInfer packs multiple short iterations with a single long iteration to fill the pipeline bubbles. When scheduling the iterations, it approximates the execution time of the Attention layer for each iteration, and decides a pattern to fully mitigate the bubbles. In the example of Figure 9b, there are two prefill iterations with input sequence length $L = 5$, one prefill iteration with input sequence length $L = 2$, and three decode iterations, i.e., $L = 1$. By similar computations when discussing KV cache swapping in §4.1, FluidInfer can obtain an estimate of their respective execution length. Then it will determine the execution pipeline greedily. It first chooses the longest iterations, which are in this case, 1 and 4 and assigns them to two parallel threads. Since there are no bubbles up till now, it will choose the next longest iteration, which is 2, and tries to assign it two the third thread. However, this lead to a Type A bubble, so it will try to choose the longest iteration left that can fill the bubble but not exceed it. In this case, we are left with 3, 5, and 6, which are all decode iterations (of approximately the same length), so it will choose 3 in the first come first serve manner. Then, it repeats the step until there are no iterations left, the bubble is filled, or all the iterations left cannot fill into the bubble while not exceeding it. If the KV cache can hold more iterations from different requests in parallel, the scheduling will return to choosing the longest iteration available and assigning it to a new thread. Iteratively, FluidInfer can achieve nearly optimal pipeline, mitigating nearly all bubbles.

5 Implementation

FluidInfer is a distributed LLM inference system with an API frontend, a scheduler, a KV cache manager, and a distributed execution engine. The API frontend is implemented with Python, FastAPI, and uvicorn. It is designed to be compatible with the OpenAI API, and implements a few popular HuggingFace transformer-based models such as the OPT series [3] and the LLaMA series [2]. The scheduler and the KV cache manager are implemented also in Python, equipped with the aforementioned techniques. The distributed execution engine is built on top of vLLM [12] and written in C++/CUDA, which is still under development. This also involves some implementation-level optimization such as fusing some kernels to adopt the proposed techniques.

6 Evaluation

In this section, I evaluate the performance of FluidInfer under a variety of workloads. Since FluidInfer is not fully implemented, I alternatively implement a simulation system to serve the evaluation purpose, which accurately computes the theoretical execution time of each step and suspends the corresponding amount of time to simulate actual execution. Memory communication, compute resources, KV cache limit, and other overheads are all taken into account.

6.1 Experimental Setup

Models and configurations. I use the OPT series models [3] with 125M, 1.3B, and 6.7B parameters for evaluation. These sizes are chosen to cover a wide range of model sizes, from small models that can be served on a single GPU to large models that require multiple GPUs. I also simulate the following GPU setups: a single NVIDIA GeForce RTX 4090 GPU (24GB), four NVIDIA GeForce RTX 4090 GPUs (96GB in total), and four NVIDIA A100 GPUs (160GB in total).

Workloads. I synthesize workloads based on the ShareGPT dataset [23], which is a collection of real-world ChatGPT conversations. For all the multi-round conversations, I only use the first round, i.e., one query to ChatGPT and one response from ChatGPT. The distribution of the ShareGPT dataset is shown in Figure 10 I further tokenize the datasets with the tokenizers of corresponding models, and filter out the requests that are longer than the maximum context length supported by the model. To simulate real-world queries, I implement a producer-consumer

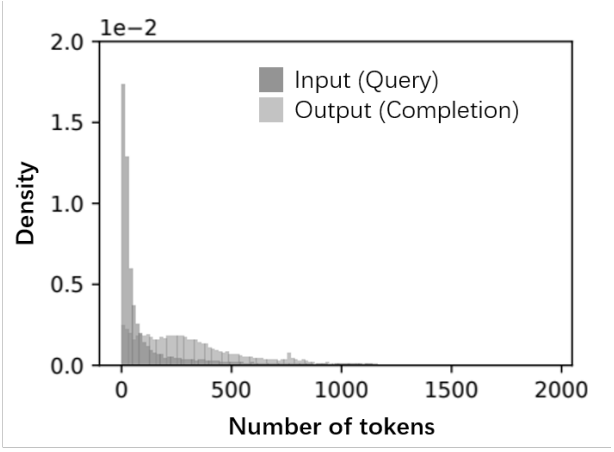

Figure 10: The distribution of input and output sequence lengths of the ShareGPT dataset [23].

model where the consumer is FluidInfer and the producer sends requests based on a Poisson process. In other words, each time the producer submits a request to the queue, it waits for a certain amount of time according to an exponential distribution with the request rate (number of requests per second) as the parameter. It is well-known that the interarrival time distribution of a Poisson process is exactly the exponential distribution with the same parameter, and Poisson processes can model random events distributed in time, i.e., users sending requests to the server.

Baselines. Orca [13], FastServe [18], and vLLM [12] are popular distributed serving systems for LLM inference, with vLLM being the current state-of-the-art solution. With the KV cache manager and the scheduler in the simulator, I can simulate the execution of these systems and compare their performance with FluidInfer.

Key metrics. *Throughput* is the primary focus of FluidInfer. It measure the number of tokens processed in a fixed amount of time. *Latency* is another critical metric in LLM inference serving since LLMs are nowadays mainly used for interactive applications such as chatbots [1, 2, 3] that require real-time responses. This can be divided in two aspects: the normalized latency which computes the average time required to generate a token, and the latency bound which represents the maximum latency between two generated tokens of a same request. In particular, throughput is essentially the same as normalized latency, in that tokens per unit time is the inverse of time per token, so we consider only normalized latency. On the other hand, latency bound is useful for guaranteeing bounded behavior in LLM inference serving, enhancing user experience.

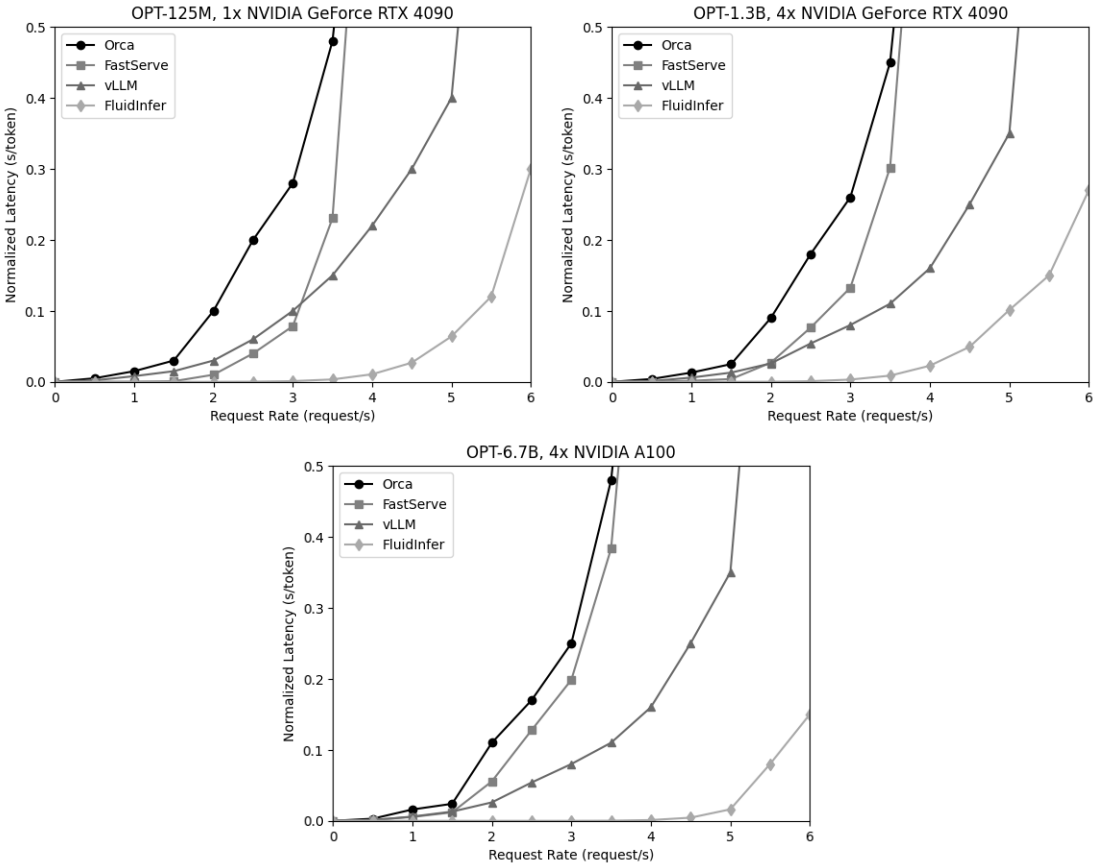

Figure 11: Normalized latency of FluidInfer and the baseline systems.

6.2 Normalized Latency

As is shown in Figure 11, FluidInfer outperforms the baseline systems in terms of normalized latency. In particular, FluidInfer achieves 2x to 4x throughput improvement over vLLM [12], and even more significant throughput improvement over Orca [13] and FastServe [18]. This is because FluidInfer overrides the KV cache limit and allows larger batch sizes, thus achieving higher throughput. The advantage of FluidInfer becomes more significant as the model size increases, because the KV cache limit becomes more severe in these cases.

6.3 Latency Bound

As is shown in Figure 12, FluidInfer also outperforms the baseline systems in terms of latency bound. In particular, FluidInfer achieves approximately 3x reduction in latency bound over vLLM [12], and even more significant reduction over Orca [13] and FastServe [18]. This is because FluidInfer packs multiple iterations in the pipeline to minimize the pipeline bubbles, thus achieving lower latency bound. The advantage of FluidInfer becomes more significant as the model size increases, because the hidden size increases and the supported context length

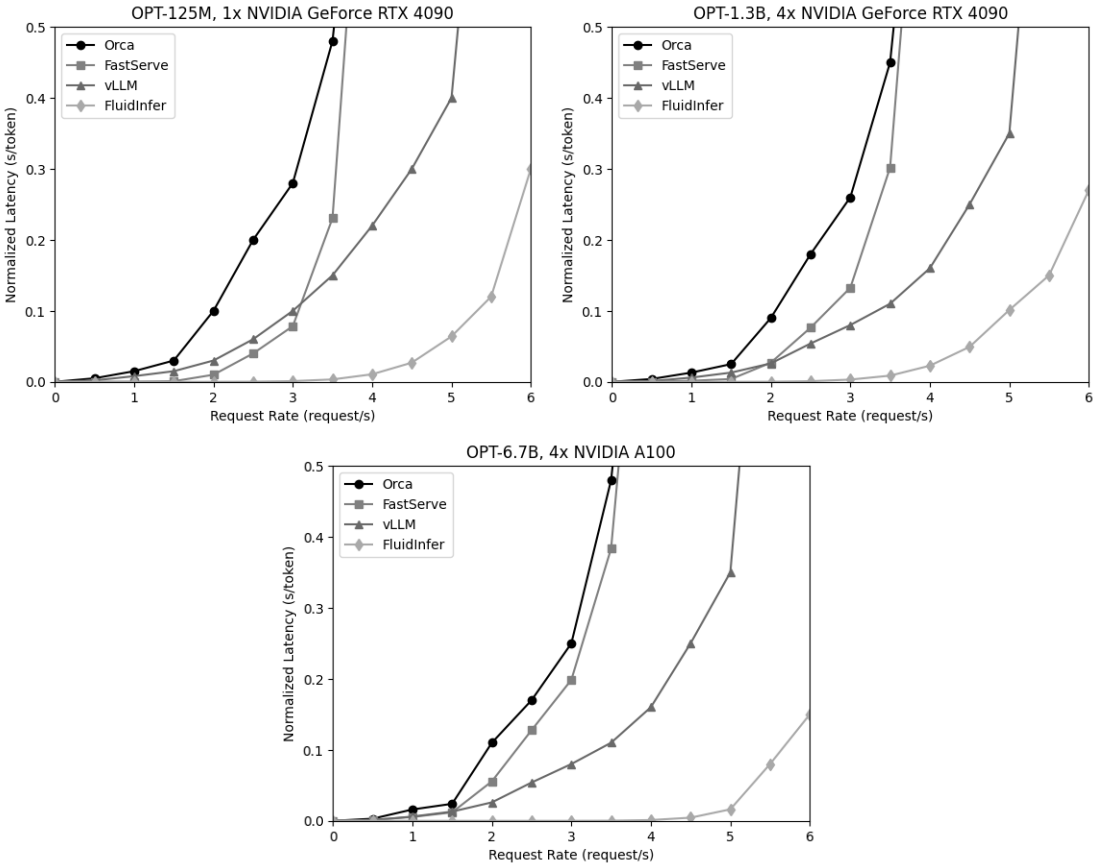

Figure 12: Latency bound of FluidInfer and the baseline systems.

is extended, leading to longer prefill iterations, more deviations in lengths, and larger pipeline bubbles.

7 Conclusion

In this paper, I have presented FluidInfer, a novel distributed serving system for the inference of large language models (LLMs) that can achieve higher throughput and lower latency than existing systems. FluidInfer addresses two major challenges in LLM inference serving: the KV cache limitation and the pipeline bubbles. To overcome the KV cache limitation, FluidInfer abandons the notion of batch size and arbitrarily concatenates or splits requests in non-Attention layers, while swapping KV cache between GPU and host memory to enable parallelized processing at much larger scales. To minimize the pipeline bubbles, FluidInfer distinguishes iterations in different phases and with different sequence lengths, and packs multiple iterations in the pipeline to fill the gaps. FluidInfer is evaluated using a comprehensive simulator under a variety of workloads, models, and configurations4. The results show that FluidInfer outperforms state-of-

the-art LLM serving systems by 2x to 4x improvement in throughput, and achieves approximately 3x lower latency bound5. FluidInfer is not fully implemented, but the simulation results demonstrate its potential and feasibility.

Future work. There are several directions for future work:

- Finish the implementation of FluidInfer. The paper only implements a simulator to evaluate the performance of FluidInfer. It would be worthwhile to finish the implementation of FluidInfer and compare its performance with the simulator to validate the simulation results.

- Extending the evaluation. The paper only evaluates end-to-end performance, which might not be sufficient to demonstrate the proficiency of each of the techniques it proposed.

- Evaluating FluidInfer on real-world LLM applications. The paper simulates the performance of FluidInfer on synthetic workloads based on the ShareGPT dataset. However, it would be more convincing to test FluidInfer on actual LLM applications such as chatbots, question answering systems, or coding assistants, and compare its efficiency and user satisfaction with existing systems.

- Extending FluidInfer to support other LLM architectures. The paper focuses on the transformer-based LLMs, which are the most popular and widely used LLMs nowadays. However, there are also other LLM architectures such as recurrent neural networks (RNNs), convolutional neural networks (CNNs), or sparse attention models , which may have different characteristics and challenges in serving. It would be interesting to explore how FluidInfer can be adapted to these architectures and whether it can achieve similar or better performance.

- Improving FluidInfer with more optimization techniques. The paper proposes two main techniques to overcome the KV cache limitation and minimize the pipeline bubbles in LLM inference serving. However, there may be other optimization techniques that can further improve the efficiency and scalability of FluidInfer, such as model compression, quantization, or pruning. It would be worthwhile to investigate how these techniques can be integrated with FluidInfer and what trade-offs they may incur.

References

[1] OpenAI, "GPT-4 Technical Report," Mar. 2023, arXiv:2303.08774 [cs]. [Online]. Available: http://arxiv.org/abs/2303.08774

[2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," Feb. 2023, arXiv:2302.13971 [cs]. [Online]. Available: http://arxiv.org/abs/2302.13971

[3] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "OPT: Open Pre-trained Transformer Language Models," Jun. 2022, arXiv:2205.01068 [cs]. [Online]. Available: http://arxiv.org/abs/2205.01068

[4] "GitHub Copilot." [Online]. Available: https://github.com/features/copilot

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374 [cs]. [Online]. Available: http://arxiv.org/abs/2107.03374

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[7] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "PaLM: Scaling Language Modeling with Pathways," Oct. 2022, arXiv:2204.02311 [cs]. [Online]. Available: http://arxiv.org/abs/2204.02311

[8] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling Laws for Neural Language Models," Jan. 2020, arXiv:2001.08361 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2001.08361

[9] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," in Proceedings of the 40th International Conference on Machine Learning, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, Jul. 2023, pp. 31 094–31 116. [Online]. Available: https://proceedings.mlr.press/v202/sheng23a.html

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," in Advances in Neural Information Processing Systems, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[11] J. Dastin and S. Nellis, "Focus: For tech giants, AI like Bing and Bard poses billion-dollar search problem," Reuters, Feb. 2023. [Online]. Available: https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/

[12] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," 2023, publisher: arXiv Version Number: 1. [Online]. Available: https://arxiv.org/abs/2309.06180

[13] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," 2022, pp. 521–538. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/yu

[14] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "TensorFlow-Serving: Flexible, High-Performance ML Serving," 2017, publisher: arXiv Version Number: 2. [Online]. Available: https://arxiv.org/abs/1712.06139

[15] N. Corporation, "Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution," 2023. [Online]. Available: https://github.com/triton-inference-server/server

[16] H. Face, "Text Generation Inference." [Online]. Available: https://huggingface.co/docs/text-generation-inference/index

[17] C. Holmes, M. Tanaka, H. Qin, M. Wyatt, A. A. Awan, and L. Kurilenko, "DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference," Nov. 2023. [Online]. Available: https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen

[18] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast Distributed Inference Serving for Large Language Models," 2023. [Online]. Available: https://arxiv.org/abs/2305.05920

[19] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills," Aug. 2023, arXiv:2308.16369 [cs]. [Online]. Available: http://arxiv.org/abs/2308.16369

[20] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d. Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[21] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in Proceedings of the 27th ACM Symposium on Operating Systems Principles, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–15. [Online]. Available: https://doi.org/10.1145/3341301.3359646

[22] J. Lages, "Transformers KV Caching Explained," Oct. 2023. [Online]. Available: https://medium.com/@joaolages/kv-caching-explained-276520203249

[23] S. Tey, "ShareGPT," Dec. 2022. [Online]. Available: https://sharegpt.com/