Efficient Large Language Model Serving with Efficient Sampler

1 Introduction

The emergence of large language models (LLMs) like GPT [] and Claude [] have enabled new applications such as programming assistants [6, 18] and universal chatbots [19, 35] that are profoundly impacting our work and daily routines. The significant GPU compute required for running inference on large language models, coupled with significant increase in their usage has made LLM inference a dominant GPU workload today. Thus, increasing the throughput and hence reducing the cost per request of *LLM serving systems* is becoming more important.

At the core of LLMs serving lies an autoregressive Transformer model generation mechanism []. This model generates words (tokens), one at a time, based on the input (prompt) and the previous sequence of the output's tokens it has generated so far. For each request, LLMs first go through the *prefill* phase to compute the whole prompt, and then iteratively generative the new tokens, which is called *decode* phase. The prefill phase is *compute-bound* due to it processes large number of prompt tokens parallel, whereas decode phase is *memory-bound* since only one token per requests is computed but the whole previous sequence needs to be accessed. These two-phase generation phases underutilizing the GPU in different ways but both limiting the serving throughput.

Improving the throughput is possible by batching multiple requests together. However, to process multiple requests in batch, the selection of different phases requests should be well scheduled. Figure 1 illustrates that prefill and decode has different batching efficiency and memory occupation. Due to the differences, batching schedules can be classified into two categories, *prefill-oriented* and *decode-oriented*. *Prefill-oriented* schedule will cause decode underutilize ALU computation bandwidth, whereas *decode-oriented* batch size settings will cause `out of memory` problem for prefill. Since prefill and decode take up the main part of serving process, the way the requests are batched is critical in determining the system throughput. When batch scheduled inefficiently, the underutilized decode and compensatory operation (such as swapping)

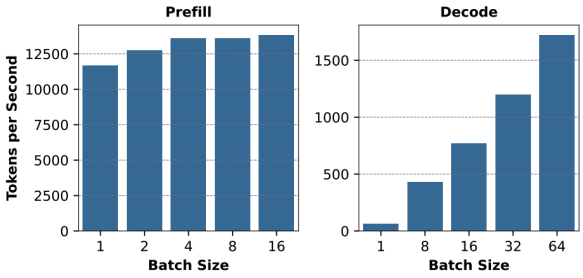

Figure 1: Dummy Picture from Sarathi-serve. Showing the batching efficiency different for prefill and decode

for prefill out of memory issues can severely degrade GPU computational efficiency, thereby devastating system throughput.

In this paper, we observe that existing LLM serving systems [] fall short of scheduling requests efficiently. This is mainly because they schedule requests batching by the *max_batch_size*, as most deep learning training systems used to compute the uniform and stable sequences. However, unlike the sequences in the traditional deep learning training workloads, the serving sequences exhibit distinctly different characteristics: it dynamically grows and shrinks at each generation(decode) step, and its lifetime and length are not known a priori. The sequences can vary a lot in length and unstable lifetime, making uniformed shaping impossible, for example in Figure **??**. These characteristics make the existing systems' approach significantly inefficient in two ways:

First, the existing systems [] suffers from under-utilization of devices bandwidth. LLM services memory usage can be divided into three main parts: model weights, KV Cache [], and activations - the ephemeral tensors allocated during LLM inference. To schedule the sequences by max_batch_size, they partition the memory by proposing an unverified sequence length corresponding to max_batch_size. This can result in severe imbalance on either activations or KV Cache, then lim-

iting the effectiveness in prefill computing or decode memory. Moreover, even if the actual length is known a priori, the verified average length still has the same problem: As the sequence length varies by requests, the system can not afford the dynamic adjustment of the memory allocation, where still exists the problem. Besides, maximizing the memory usage also causes significant extra cost, since compensatory operation is frequent called to prevent from memory overflow. Actually, our profiling results shown that only xxx% and xxx% of the KV Cache and activations is used by actual tokens.

Second, the existing systems fail to exploit the opportunities of memory beyond KV Cache. During *decoding* phase, compensatory operation (CPU `swapping`) is often inevitably adopted due to growing sequences. Moreover, LLM services often arm with advanced techniques, such as multi-LoRA serving [] and prompt caching [], that consumes large amount of memory. In these scenarios, systems can leverage CPU memory through *swapping* to enable larger available capacity on accelerators, thereby boosting performance. However, leveraging CPU memory is not possible in the existing systems because one-off swapping (as Figure **??**) by CPU-accelerator communication introduces additional overhead, due to its large amount and low efficiency.

To address the above limitations, we propose Efficient Sampler. We optimize the Sampler part by two menas: 1. intra-micro-pipeling, and 2. inter-logits parallel. With these two methods, we can largely alleviate the Sampler memory within a certain level, enabling as much as memory to be used by KV Cache. Along with the enlarged KV Cache, we co-design the ElasticBatch scheduler, which batches the requests only by token capacity rather than max_batch_size, which presents varying batch size and thus called *elastic batching*. Token capacity can directly deciding the batching efficiency of prefill, guaranteeing the stable high throughput. Also, decode's batch_size constraint is also unleashed for maximizing potential throughput. StreamSwap conduct the CPU-accelerator offload through chunked communication instead of traditional one-off []. When the LLM serving calls KV Cache swapping, StreamSwap chunks the offloaded KV Cache by layers and communicate the corresponding a head of next layer computation, to perform a near-zero-cost offloading.

To support these two mechanism, we also design and implement the Global Memory Manager (GMM). In the initialization, GMM searches for the optimal token capacity to partition KV Cache and activation memory. During serving, the activation memory is: 1) decreased peak usage by GMM to optimize activation flow; 2) predicted by the actual usage for scheduling. Since the activation is dynamically allocated and often over-provisioned, GMM applies for a KVCache-like memory within the activation memory. This memory can be further used as a fast-swap cache for CPU-accelerator offloading. Given that StreamSwap only suffers from the first swapping chunk, StreamSwap can pre-swap the first chunk to this cache. Since same accelerators copy is extremely fast, serving system is able to enable leveraging CPU memory to accelerate system performance by this Multi-Layer swap cache and StreamSwap.

On top of these three parts, we propose *MAXGen*, a high-throughput LLM distributed serving engine that achieves maximum throughput and near-zero overhead. MAXGen supports popular LLMs such as GPT, OPT, and LLaMA with varying sizes, including the ones exceeding the memory capacity of a single GPU. Beyond this, MAXGen aims at general adaptability with multiple advanced serving techniques, such as *model parallelism* including *TP* and *PP*, *Lookahead Decoding*, and as mentioned before, multi-LoRA and prompt cache. For better collaboration, we also develop corresponding co-designed algorithms and application. Our evaluations on various models and workloads show that MAXGen improves the LLM serving throughput by ... compared to the existing systems, without affecting the model accuracy at all. The improvements are more pronounced with larger models and more complex serving techniques. In summary, we make the following contributions:

- We identify the challenges in batch scheduling and large batch serving in LLMs and quantify their impact on serving performance.
- We propose ElasticBatch and StreamSwap, a batching schedule mechanism that operates on Token Capacity and a streaming offload mechanism with data chunking.
- We design and implement Global Memory Manager, which monitors, partitions, and exploits the global memory of accelerators and CPU, propose Multi-Layer Swap Cache, to better support core mechanisms.
- We design and implement MAXGen, a distributed LLM serving engine built on top of ElasticBatch and StreamSwap.
- We evaluate MAXGen on various scenarios and demonstrate that it substantially outperforms the previous state-of-the-art solutions such as Orca and vLLM.

2 Background

2.1 LLM & Autoregressive Generation

Its *pre-fill* has a computation complexity of $24bsh^2 + 4bs^2h$, its decode has a computation complexity of $24bh^2 + 4bs'h$. For large LLM, whose $b \geq 10^5$, the decode can be considered always:

$$T_{decode} = \frac{1}{s} T_{prefill}$$

For this, an extra part is induced *in-fill*, which complexity is $24bs_ih^2 + 4bs_i^2h$, the decode becomes $24bh^2 + 4bs_i'h$. Still, if the $s + s_i \leq 10^5(h)$, the relation will still be:

$$T_{decode} = \frac{1}{s} T_{prefill} = \frac{1}{s_i} T_{infill}$$

2.2 Batching Techniques for LLMs

inflight batching, continuous batching

3 Method

3.1 Efficient Sampler

3.2 Paralleled Sampler

3.3 ElasticBatch Scheduler

not fix batch size and exploit the memory

3.4 Swapping

how to schedule, and infill operation **Multi-batch Inference** Since both *fill* and decode has strong serialization dependency, no interleaving can be done in single batch. Hence, multi-batch provides opportunity.

Multi-batch enables the $micro\_batch > 1$. Therefore, the $c = m \cdot C, m \leq B$, $m \cdot b = B$, b is the current batch size, B is the overall batch size.

When the multi-batch must have different sequence length, to solve this problem, we will also adopt Sequence Parallel to reduce the step-step bubble in filling phase.

We will chunk the step of different batch into the same T_{step} to reduce the bubble.

For Pre-fill,

First, we consider an ideal case, where multiple batches share the same sequence length. In this case, we just chunk them into the same size, the chunk $c = argmin(T_{step} + T_{bubble})$.

Then, we consider multiple batches with different sequence length. We will divide all batches into c, which will reach the $c = argmin(T_{step} + T_{bubble})$. Because the chunk will be no more than c, therefore, no bubbles. $T_{step} = T_{kernel\_util} \cdot C$

For Decode,

we need to ensure the running batches more than pp_stage. If $b < pp\_stage$, step-step bubble will happen. When $b > pp\_stage$, we can fuse batches to enlarge the kernel computation to increase the utilization, but still make sure it is great, we can fuse the batch size to times of pp_stage, and when some is finished, we open the bundle.

When In-fill,

we will chunk the in-fill length, fuse all-batches to create $b \geq pp\_stage$. Therefore, the key points is not let the $c < pp\_stage$, and then **it's the matter with the schedule method** .

Multi-batch Schedule

Have a streaming monitor, a buffer stack to keep the current operation serial. For each gpu, fill and decode task will be pushed into the buffer stack continuously. By changing the ranking strategy of the stack, we can design multiple scheduling strategy.

4 Implementation

5 Evaluation

6 Ablation Study

7 Discussion

8 Related Work

9 Conclusion

References

Notes