Taming Throughput-Latency Tradeoff in LLM Inference with *Sarathi-Serve*

Amey Agrawal, *Georgia Institute of Technology;* Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, and Bhargav Gulavani, *Microsoft Research India;* Alexey Tumanov, *Georgia Institute of Technology;* Ramachandran Ramjee, *Microsoft Research India*

https://www.usenix.org/conference/osdi24/presentation/agrawal

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Taming Throughput-Latency Tradeoff in LLM Inference with *Sarathi-Serve*

Amey Agrawal*[2], Nitin Kedia[1], Ashish Panwar[1], Jayashree Mohan[1], Nipun Kwatra[1], Bhargav S. Gulavani[1], Alexey Tumanov[2], and Ramachandran Ramjee[1]

[1]Microsoft Research India
[2]Georgia Institute of Technology

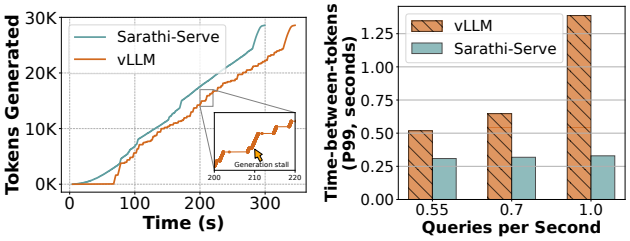
Abstract

Each LLM serving request goes through two phases. The first is *prefill* which processes the entire input prompt and produces the first output token and the second is *decode* which generates the rest of output tokens, one-at-a-time. Prefill iterations have high latency but saturate GPU compute due to parallel processing of the input prompt. In contrast, decode iterations have low latency but also low compute utilization because a decode iteration processes only a single token per request. This makes batching highly effective for decodes and consequently for overall throughput. However, batching multiple requests leads to an interleaving of prefill and decode iterations which makes it challenging to achieve both high throughput and low latency.

We introduce an efficient LLM inference scheduler, Sarathi-Serve, to address this throughput-latency tradeoff. Sarathi-Serve introduces *chunked-prefills* which splits a prefill request into near equal sized chunks and creates *stall-free* schedules that adds new requests in a batch without pausing ongoing decodes. Stall-free scheduling unlocks the opportunity to improve throughput with large batch sizes while minimizing the effect of batching on latency. Furthermore, uniform batches in Sarathi-Serve ameliorate the imbalance between iterations, resulting in minimal pipeline bubbles.

Our techniques yield significant improvements in inference performance across models and hardware under tail latency constraints. For Mistral-7B on single A100 GPUs, we achieve 2.6× higher serving capacity and up to 3.7× higher serving capacity for the Yi-34B model on two A100 GPUs as compared to vLLM. When used with pipeline parallelism on Falcon-180B, Sarathi-Serve provides up to 5.6× gain in the end-to-end serving capacity. The source code for Sarathi-Serve is available at https://github.com/microsoft/sarathi-serve.

*Part of this work was done during an internship at MSR India.



(a) Generation stall.

(b) High tail latency.

Figure 1: Yi-34B running on two A100 GPUs serving 128 requests from *arxiv-summarisation* trace. 1a highlights one of the many generation stalls lasting over several seconds in vLLM [53]. 1b shows the impact of increasing load on tail latency. Sarathi-Serve improves throughput while eliminating generation stalls.

1 Introduction

Large language models (LLMs) [34,35,52,57,71] have shown impressive abilities in a wide variety of tasks spanning natural language processing, question answering, code generation, etc. This has led to tremendous increase in their usage across many applications such as chatbots [2, 5, 6, 57], search [4, 9, 11, 19, 25], code assistants [1, 8, 20], etc. The significant GPU compute required for running inference on large models, coupled with significant increase in their usage, has made LLM inference a dominant GPU workload today. Thus, optimizing LLM inference has been a key focus for many recent systems [29, 53, 58, 59, 63, 75, 77].

Optimizing throughput and latency are both important objectives in LLM inference since the former helps keep serving costs tractable while the latter is necessary to meet application requirements. In this paper, we show that current LLM serving systems have to face a tradeoff between throughput and latency. In particular, LLM inference throughput can be increased significantly with batching. However, the way existing systems batch multiple requests leads to a compromise on either throughput or latency. For example, Figure 1b shows

that increasing load can significantly increase tail latency in a state-of-the-art LLM serving system vLLM [53].

Each LLM inference request goes through two phases – a *prefill* phase followed by a *decode* phase. The *prefill* phase corresponds to the processing of the input prompt and the *decode* phase corresponds to the autoregressive token generation. The prefill phase is compute-bound because it processes all tokens of an input prompt in parallel whereas the decode phase is memory-bound because it processes only one token per-request at a time. Therefore, decodes benefit significantly from batching because larger batches can use GPUs more efficiently whereas prefills do not benefit from batching.

Current LLM inference schedulers can be broadly classified into two categories[1], namely, *prefill-prioritizing* and *decode-prioritizing* depending on how they schedule the prefill and decode phases while batching requests. In this paper, we argue that both strategies have fundamental pitfalls that make them unsuitable for serving online inference (see Figure 2).

Traditional request-level batching systems such as FasterTransformer [7] employ *decode-prioritizing* scheduling. These systems submit a batch of requests to the execution engine that first computes the prefill phase of all requests and then schedules their decode phase. The batch completes only after all requests in it have finished their decode phase i.e., new prefills are not scheduled as long as one or more requests are doing decodes. This strategy optimizes inference for latency metric time-between-tokens or TBT – an important performance metric for LLMs. This is because new requests do not affect the execution of ongoing requests in their decode phase. However, *decode-prioritizing* schedulers severely compromise on throughput because even if some requests in a batch finish early, the execution continues with reduced batch size until the completion of the last request.

Orca [75] introduced iteration-level batching wherein requests can dynamically enter or exit a batch at the granularity of individual iterations. Iteration-level batching improves throughput by avoiding inefficiencies of request-level batching systems. Orca and several other recent systems like vLLM [23] combine iteration-level batching with *prefill-prioritizing* scheduling wherein they eagerly schedule the prefill phase of one or more requests first i.e., whenever GPU memory becomes available. This way, *prefill-prioritizing* schedulers have better throughput because computing prefills first allows subsequent decodes to operate at high batch sizes. However, prioritizing prefills leads to high latency because it interferes with ongoing decodes. Since prefills can take arbitrarily long time depending on the lengths of the given prompts, *prefill-prioritizing* schedulers lead to an undesirable phenomenon that we refer to as *generation stalls* in this paper. For example, Figure 1a shows that a generation stall in vLLM can last over several seconds.

Another challenge introduced by traditional iteration-level scheduling systems like Orca [75] is pipeline stalls or bubbles [49]. These appear in pipeline-parallelism (PP) deployments that are needed to scale LLM inference across several nodes. In servers with high bandwidth connectivity such as NVIDIA DGX A100 [16], tensor-parallelism (TP) [64] can enable deployment of an LLM on up to 8 GPUs, supporting large batch sizes with low latencies. However, TP can have prohibitively high latencies when hyper-clusters are unavailable [33]. Thus, as an alternative to TP, pipeline-parallelism (PP) [33, 55] is typically used across commodity networks. Existing systems rely on micro-batches to mitigate pipeline stalls or bubbles [49]. However, the standard micro-batch based scheduling can still lead to pipeline bubbles due to the unique characteristics of LLM inference. Specifically, LLM inference consists of a mixture of varying length prefills and decodes. The resulting schedule can thus have wildly varying runtimes across different micro-batches that waste GPU cycles and degrade the overall system throughput.

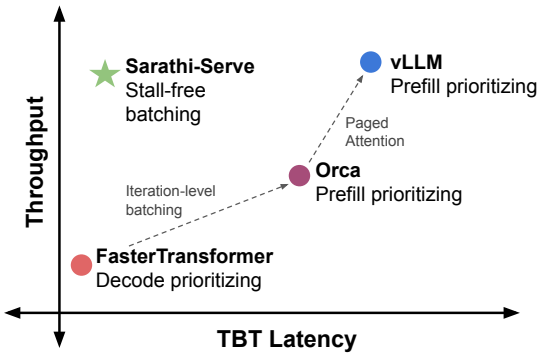


Figure 2: Current LLM serving systems involve a tradeoff between throughput and latency depending on their scheduling policy. Prioritizing prefills optimizes throughput but sacrifices TBT (time-between-tokens) tail latency whereas prioritizing decodes has the opposite effect. Sarathi-Serve serves high throughput with low TBT latency via stall-free batching. (The figure is illustrative and actual values will depend on the model and workload characteristics.)

To address these challenges, we propose Sarathi-Serve, a scheduler to balance the throughput-latency tradeoff for scalable online LLM inference serving. Sarathi-Serve is based on two key ideas: *chunked-prefills* and *stall-free* scheduling. *Chunked-prefills* splits a prefill request into equal compute-sized chunks and computes a prompt's prefill phase over multiple iterations (each with a subset of the prompt tokens). *Stall-free* scheduling allows *new requests to join a running batch without pausing ongoing decodes*. This involves constructing a batch by coalescing all the on-going decodes with one (or more) prefill chunks from new requests such that each batch reaches the pre-configured chunk size. Sarathi-Serve builds upon iteration-level batching but with an important distinction: it throttles the number of prefill tokens in each it-

[1]We classify recent schedulers Splitwise [58] and DistServe [77] under a third category "disaggregated" and discuss them in §6.

eration while admitting new requests in a running batch. This not only bounds the latency of each iteration, but also makes it nearly independent of the total length of input prompts. This way, Sarathi-Serve minimizes the effect of computing new prefills on the TBT of ongoing decodes enabling both high throughput and low TBT latency.

In addition, hybrid batches (consisting of prefill and decode tokens) constructed by Sarathi-Serve have a near-uniform compute requirement. With pipeline-parallelism, this allows us to create balanced micro-batching based schedules that significantly reduce pipeline bubbles and improve GPU utilization, thus allowing efficient and scalable deployments.

We evaluate Sarathi-Serve across different models and hardware — Mistral-7B on a single A100, Yi-34B on 2 A100 GPUs with 2-way tensor parallelism, LLaMA2-70B on 8 A40 GPUs, and Falcon-180B with 2-way pipeline and 4-way tensor parallelism across 8 A100 GPUs connected over commodity ethernet. For Yi-34B, Sarathi-Serve improves system serving capacity by up to 3.7× under different SLO targets. Similarly for Mistral-7B, we achieve up to 2.6× higher serving capacity. Sarathi-Serve also reduces pipeline bubbles, resulting in up to 5.6× gains in end-to-end serving capacity for Falcon-180B deployed with pipeline parallelism.

The main contributions of our paper include:

1. We identify a number of pitfalls in the current LLM serving systems, particularly in the context of navigating the throughput-latency tradeoff.
2. We introduce two simple-yet-effective techniques, *chunked-prefills* and *stall-free batching*, to improve the performance of an LLM serving system.
3. We show generality through extensive evaluation over multiple models, hardware, and parallelism strategies demonstrating that Sarathi-Serve improves model serving capacity by up to an order of magnitude.

2 Background

In this section, we describe the typical LLM model architecture along with their auto-regressive inference process. We also provide an overview of the scheduling policies and important performance metrics.

2.1 The Transformer Architecture

Popular large language models, like, GPT-3 [18], LLaMA [66], Yi [24] etc. are decoder-only transformer models trained on next token prediction tasks. These models consist of a stack of layers identical in structure. Each layer contains two modules – self-attention and feed-forward network (FFN).

Self-attention module: The self-attention module is central to the transformer architecture [67], enabling each part of a sequence to consider all previous parts for generating a contextual representation. During the computation of self-attention, first the Query (Q), Key (K) and Value (V) vectors corresponding to each input token are obtained via a linear transformation. Next, the *attention* operator computes a semantic relationship among all tokens of a sequence. This involves computing a dot-product of each Q vector with K vectors of all preceding tokens of the sequence, followed by a softmax operation to obtain a weight vector, which is then used to compute a weighted average of the V vectors. This attention computation can be performed across multiple *heads*, whose outputs are combined using a linear transformation.

Feed-forward network (FFN): FFN typically consists of two linear transformations with a non-linear activation in between. The first linear layer transforms an input token embedding of dimension h to a higher dimension $h2$. This is followed by an activation function, typically ReLU or GELU [27,46]. Finally, the second linear layer, transforms the token embedding back to the original dimension h.

2.2 LLM Inference Process

Autoregressive decoding: LLM inference consists of two distinct phases – a *prefill* phase followed by a *decode* phase. The prefill phase processes the user's input prompt and produces the first output token. Subsequently, the decode phase generates output tokens one at a time wherein the token generated in the previous step is passed through the model to generate the next token until a special *end-of-sequence* token is generated. Note that the decode phase requires access to all the keys and values associated with all the previously processed tokens to perform the attention operation. To avoid repeated recomputation, contemporary LLM inference systems store activations in KV-cache [7, 64, 75].

A typical LLM prompt contains 100s-1000s of input tokens Table 2, [76]. During the prefill phase all these prompt tokens are processed in parallel in a single iteration. The parallel processing allows efficient utilization of GPU compute. On the contrary, the decode phase involves a full forward pass of the model over a single token generated in the previous iteration. This leads to low compute utilization making decodes memory-bound.

Batched LLM inference in multi-tenant environment: A production serving system must deal with concurrent requests from multiple users. Naively processing requests in a sequential manner leads to a severe under-utilization of GPU compute. In order to achieve higher GPU utilization, LLM serving systems leverage batching to process multiple requests concurrently. This is particularly effective for the decode phase processing which has lower computational intensity at low batch sizes. Higher batch sizes allows the cost of fetching model parameters to be amortized across multiple requests.

Recently, several complementary techniques have been proposed to optimize throughput by enabling support for larger batch sizes. Kwon et al. propose PagedAttention [53], which allows more requests to concurrently execute, eliminating fragmentation in *KV-cache*. The use of Multi Query Attention

Algorithm 1 Request-level batching. New requests are admitted only if there are no decodes left (line 3). This optimizes TBT but wastes GPU compute in many decode-only iterations (line 10) with potentially small batch sizes.

```
1:  Initialize current batch B ← ∅
2:  while True do
3:      if B = ∅ then
4:          R_new ← get_next_request()
5:          while can_allocate_request(R_new) do
6:              B ← B + R_new
7:              R_new ← get_next_request()
8:          prefill(B)
9:      else
10:         decode(B)
11:         B ← filter_finished_requests(B)
```

Algorithm 2 Iteration-level batching (vLLM). Prefills are executed eagerly (lines 8-9), potentially introducing a generation stall for ongoing decodes (line 12).

```
1:  Initialize current batch B ← ∅
2:  while True do
3:      B_new ← ∅
4:      R_new ← get_next_request()
5:      while can_allocate_request(R_new) do
6:          B_new ← B_new + R_new
7:          R_new ← get_next_request()
8:      if B_new ≠ ∅ then
9:          prefill(B_new)
10:         B ← B + B_new
11:     else
12:         decode(B)
13:     B ← filter_finished_requests(B)
```

(MQA) [61], Group Query Attention (GQA) [30] in leading edge LLM models like LLaMA2 [66], Falcon [31] and Yi [24] has also significantly helped in alleviating memory bottleneck in LLM inference. For instance, LLaMA2-70B model has a 8× smaller KV-cache footprint compared to LLaMA-65B.

2.3 Multi-GPU LLM Inference

With ever-increasing growth in model sizes, it becomes necessary to scale LLMs to multi-GPU or even multi-node deployments [22, 59]. Furthermore, LLM inference throughput, specifically that of the decode phase is limited by the maximum batch size we can fit on a GPU. Inference efficiency can therefore benefit from model-parallelism which allows larger batch sizes by sharding model weights across multiple GPUs. Prior work has employed both tensor-parallelism (TP) [64] and pipeline-parallelism (PP) [7, 72, 75] for this purpose.

TP shards each layer across the participating GPUs by splitting the model weights and KV-cache equally across GPU workers. This way, TP can linearly scale per-GPU batch size. However, TP involves a high communication cost due to two all-reduce operations per layer – one in attention computation and the other in FFN [64]. Moreover, since these communication operations are in the critical path, TP is preferred only within a single node where GPUs are connected via high bandwidth interconnects like NVLink.

Compared to TP, PP splits a model layer-wise, where each GPU is responsible for a subset of layers. To keep all GPUs in the 'pipeline' busy, *micro-batching* is employed. These micro-batches move along the pipeline from one stage to the next at each iteration. PP has much better compute-communication ratio compared to TP, as it only needs to send activations once for multiple layers of compute. Furthermore, PP requires communication only via point-to-point communication operations, compared to the more expensive allreduces in TP. Thus, PP is more efficient than TP when high-bandwidth interconnects are unavailable *e.g.,* in cross-node deployments.

2.4 Performance Metrics

There are two primary latency metrics of interest for LLM serving: TTFT (time-to-first-token) and TBT (time-between-tokens). For a given request, TTFT measures the latency of generating the first output token from the moment a request arrives in the system. This metric reflects the initial responsiveness of the model. TBT on the other hand measures the interval between the generation of consecutive output tokens of a request, and affects the overall perceived fluidity of the response. When system is under load, low throughput can lead to large scheduling delays and consequently higher TTFT.

In addition, we use a throughput metric, *Capacity*, defined as the maximum request load (queries-per-second) a system can sustain while meeting certain latency targets. Higher capacity is desirable because it reduces the cost of serving.

2.5 Scheduling Policies for LLM Inference

The scheduler is responsible for admission control and batching policy. For the ease of exposition, we investigate existing LLM inference schedulers by broadly classifying them under two categories – *prefill-prioritizing* and *decode-prioritizing*.

Conventional inference engines like FasterTransformer [7], Triton Inference Server [17] use *decode-prioritizing* schedules with request-level batching *i.e.,* they pick a batch of requests and execute it until *all* requests in the batch complete (Algorithm 1). This approach reduces the operational complexity of the scheduling framework but at the expense of inefficient resource utilization. Different requests in a batch typically have a large variation in the number of input and output tokens. Request-level schedulers pad shorter requests with zeros to match their length with the longest request in the batch which results in wasteful compute and longer wait times for pending requests [75].

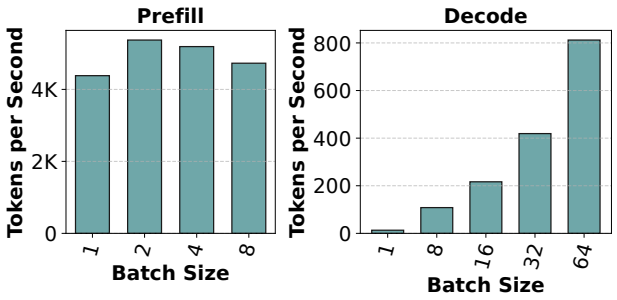


Figure 3: Throughput of the prefill and decode phases with different batch sizes for Mistral-7B running on a single A100 GPU. We use prompt length of 1024 for both prefill and decode experiments. Note that different y-axis, showing prefills are much more efficient than decode. Further, note that *batching boosts decode throughput almost linearly but has a marginal effect on prefill throughput.*

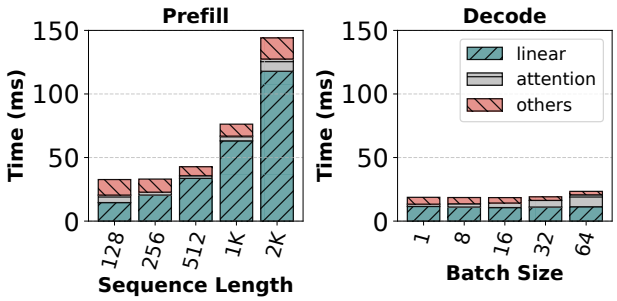


Figure 4: Prefill and decode time with different input sizes for Mistral-7B running on single A100 GPU. Linear layers contribute to the majority of runtime in both prefill and decode phases. Due to the low arithmetic intensity in decode batches, the cost of linear operation for 1 decode token is nearly same as 128 prefill tokens.

To avoid wasted compute of request-level batching, Orca [75] introduced a fine-grained iteration-level batching mechanism where requests can dynamically enter and exit a batch after each model iteration.(Algorithm 2). This approach can significantly increase system throughput and is being used in many LLM inference serving systems today *e.g.,* vLLM [23], TensorRT-LLM [21], and LightLLM [12].

Current iteration-level batching systems such as vLLM [23] and Orca [75] use *prefill-prioritizing* schedules that eagerly admit new requests in a running batch at the first available opportunity, e.g., whenever GPU memory becomes available. Prioritizing prefills can improve throughput because it increases the batch size of subsequent decode iterations.

3 Motivation

In this section, we first analyse the cost of prefill and decode operations. We then highlight the throughput-latency trade-off and pipeline bubbles that appear in serving LLMs.

3.1 Cost Analysis of Prefill and Decode

As discussed in §2.2, while the *prefill* phase processes all input tokens in parallel and effectively saturates GPU compute, the *decode* phase processes only a single token at a time and is very inefficient. Figure 3 illustrates throughput as a function of batch size, and we can observe that while for decode iterations throughput increases roughly linearly with batch size, prefill throughput almost saturates even with a single request.

Takeaway-1: *The two phases of LLM inference – prefill and decode – demonstrate contrasting behaviors wherein batching boosts decode phase throughput immensely but has little effect on prefill throughput.*

Figure 4 breaks down the prefill and decode compute times into linear, attention and others, and shows their individual contributions. From the figure, we see that linear operators contribute to the majority of the runtime cost. While attention cost grows quadratically with sequence length, linear operators still contribute more than 80% to the total time even at high sequence lengths. Therefore, optimizing linear operators is important for improving LLM inference.

Low Compute Utilization during Decodes: Low compute utilization during the decode phase is a waste of GPU's processing capacity. To understand this further, we analyze the arithmetic intensity of prefill and decode iterations. Since the majority of the time in LLM inference is spent in linear operators, we focus our analysis on them.

Matrix multiplication kernels overlap memory accesses along with computation of math operations. The total execution time of an operation can be approximated to $T = \max(T_{\text{math}}, T_{\text{mem}})$, where T_{math} and T_{mem} represent the time spent on math and memory fetch operations respectively. An operation is considered memory-bound if $T_{\text{math}} < T_{\text{mem}}$. Memory-bound operations have low Model FLOPs Utilization (MFU) [35]. On the other hand, compute-bound operations have low Model Bandwidth Utilization (MBU). When $T_{\text{math}} = T_{\text{mem}}$, both compute and memory bandwidth utilization are maximized. Arithmetic intensity quantifies the number of math operations performed per byte of data fetched from the memory. At the optimal point, the arithmetic intensity of operation matches the FLOPS-to-Bandwidth ratio of the device. Figure 5 shows arithmetic intensity as a function of the number of tokens in the batch for linear layers in LLaMA2-70B running on four A100 GPUs. Prefill batches amortize the cost of fetching weights of the linear operators from HBM memory to GPU cache over a large number of tokens, allowing it to have high arithmetic intensity. In contrast, decode batches have very low computation intensity. Figure 6 shows the total execution time of linear operators in

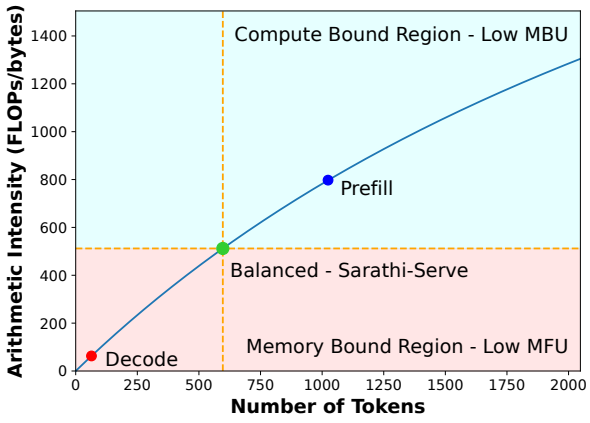


Figure 5: Arithmetic intensity trend for LLaMA2-70B linear operations with different number of token running on four A100s. Decode batches have low arithmetic intensity *i.e.,* they are bottlenecked by memory fetch time, leading to low compute utilization. Prefill batches are compute bound with sub-optimal bandwidth utilization. Sarathi-Serve forms balanced batches by combining decodes and prefill chunks to maximize both compute and bandwidth utilization.

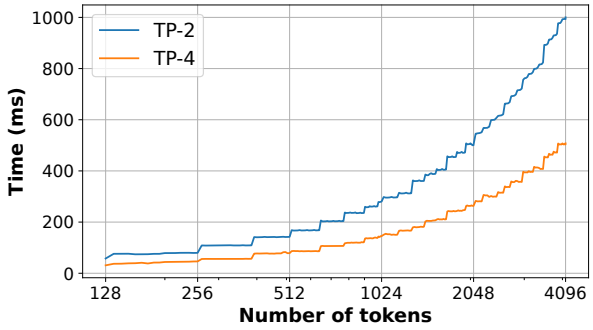


Figure 6: Linear layer execution time as function of number of tokens in a batch for LLaMA2-70B on A100(s) with different tensor parallel degrees. When the number of tokens is small, execution time is dictated by the cost of fetching weights from HBM memory. Hence, execution time is largely stagnant in the 128-512 tokens range, especially for higher tensor parallel degrees. Once the number of tokens in the batch cross a critical threshold, the operation become compute bound and the runtime increases linearly with number of tokens.

an iteration for LLaMA2-70B as a function of the number of tokens. Note that execution time increases only marginally in the beginning *i.e.,* as long as the batch is in a memory-bound regime, but linearly afterwards *i.e.,* when the batch becomes compute-bound.[2]

Takeaway-2: *Decode batches operate in memory-bound regime leaving compute underutilized. This implies that more tokens can be processed along with a decode batch without significantly increasing its latency.*

3.2 Throughput-Latency Trade-off

Iteration-level batching improves system throughput but we show that it comes at the cost of high TBT latency due to a phenomenon we call *generation stalls*.

Figure 7 compares different scheduling policies. The example shows a timeline (left to right) of requests A, B, C and D. Requests A and B are in decode phase at the start of the interval and after one iteration, requests C and D enter the system. Orca and vLLM both use FCFS iteration-level batching with eager admission of prefill requests but differ in their batch composition policy. Orca supports hybrid batches composed of both prefill and decode requests whereas vLLM only supports batches that contain either all prefill or all decode requests. Irrespective of this difference, both Orca and vLLM can improve throughput by maximizing the batch size

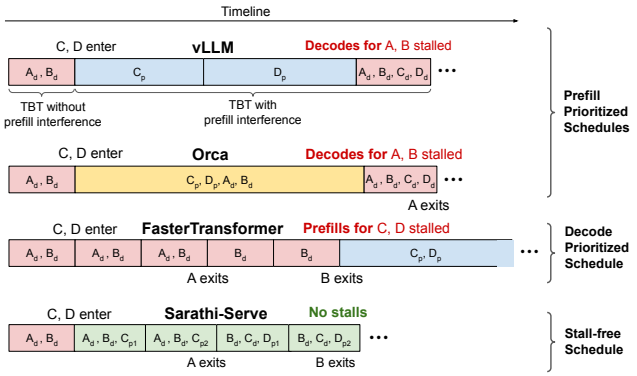


Figure 7: A generation stall occurs when one or more prefills are scheduled in between consecutive decode iterations of a request. A, B, C and D represent different requests. Subscript d represents a decode iteration, p represents a full prefill and $p0$, $p1$ represent two chunked prefills of a given prompt. vLLM induces generation stalls by scheduling as many prefills as possible before resuming ongoing decodes. Despite supporting hybrid batches, Orca cannot mitigate generation stalls because the execution time of batches containing long prompts remains high. FasterTransformer is free of generation stalls as it finishes all ongoing decodes before scheduling a new prefill but compromises on throughput due to low decode batch size. In contrast, Sarathi-Serve generates a schedule that eliminates generation stalls yet delivers high throughput.

in subsequent decode iterations. However, eagerly scheduling prefills of requests C and D delays the decodes of already running requests A and B because an iteration that computes one or more prefills can take several seconds depending on the

[2]Theoretically, we expect the operators to become compute-bound at ∼200 tokens on A100 GPUs, however, in practice we observe that it happens at ∼500-600 tokens for higher tensor parallel dimensions due to fixed overheads.

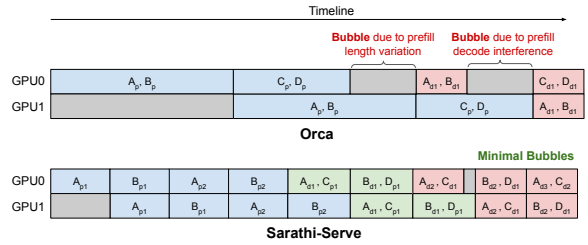


Figure 8: A 2-way pipeline parallel iteration-level schedule in Orca across 4 requests (A,B,C,D) shows the existence of pipeline bubbles due to non-uniform batch execution times. Sarathi-Serve is able to minimize these stalls by creating uniform-compute batches.

lengths of input prompts. Therefore, *prefill-prioritizing* schedulers can introduce *generation stalls* for ongoing decodes resulting in latency spikes caused by high TBT.

In contrast to iteration-level batching, request-level batching systems such as FasterTransformer [7] do not schedule new requests until *all* the already running requests complete their decode phase (line 3 in Algorithm 1). In Figure 7, the prefills for requests C and D get stalled until requests A and B both exit the system. Therefore, *decode-prioritizing* systems provide low TBT latency albeit at the cost of low system throughput. For example, Kwon et al. [53] show that iteration-level batching with PagedAttention can achieve an order of magnitude higher throughput compared to FasterTransformer.

One way to reduce latency spikes in iteration-level batching systems is to use smaller batch sizes as recommended in Orca [75]. However, lowering batch size adversely impacts throughput as shown in §2.2. Therefore, existing systems are forced to trade-off between throughput and latency depending on the desired SLOs.

Takeaway-3: *The interleaving of prefills and decodes involves a trade-off between throughput and latency for current LLM inference schedulers. State-of-the-art systems today use prefill-prioritizing schedules that trade TBT latency for high throughput.*

3.3 Pipeline Bubbles waste GPU Cycles

Pipeline-parallelism (PP) is a popular strategy for cross-node deployment of large models, owing to its lower communication overheads compared to Tensor Parallelism (TP). A challenge with PP, however, is that it introduces *pipeline bubbles* or periods of GPU inactivity as subsequent pipeline stages have to wait for the completion of the corresponding micro-batch in the prior stages. Pipeline bubbles is a known problem in training jobs, where they arise between the forward and backward passes due to prior stages needing to wait for the backward pass to arrive. Micro-batching is a common technique used in PP training jobs to mitigate pipeline bubbles [33, 49, 55].

Inference jobs only require forward computation and therefore one might expect that micro-batching can eliminate pipeline bubbles during inference. In fact, prior work on transformer inference, such as, FasterTransformer [7] and FastServe [72] use micro-batches but do not mention pipeline-bubbles. Recently proposed Orca [75] also suggests that iteration-level scheduling eliminates bubbles in pipeline scheduling (see Figure 8 in [75]). However, our experiments show that even with iteration-level scheduling, pipeline bubbles can waste significant GPU cycles with PP (§5.3).

Each micro-batch (or iteration) in LLM inference can require a different amount of compute (and consequently has varying execution time), depending on the composition of prefill and decode tokens in the micro-batch (see Figure 8). We identify three types of bubbles during inference: (1) bubbles like PB_1 that occur due to the varying number of prefill tokens in two consecutive micro-batches (2) bubbles like PB_2 that occur due to different compute times of prefill and decode stages when one is followed by the other, and (3) bubbles like PB_3 that occur due to difference in decode compute times between micro-batches since the attention cost depends on the accumulated context length (size of the KV-cache) and varies across requests. For Falcon-180B, a single prompt of 4k tokens takes $\approx$ 1150 ms to execute compared to a decode only iteration with batch size 32 which would take about $\approx$ 200 ms to execute. Interleaving of these iteration could result in a bubble of $\approx$ 950 ms. These pipeline bubbles are wasted GPU cycles and directly correspond to a loss in serving throughput and increased latency. This problem is aggravated with increase in prompt lengths and batch size, due to longer and more frequent prefill iterations respectively. If we can ensure that each micro-batch performs uniform computation, we can mitigate these pipeline bubbles.

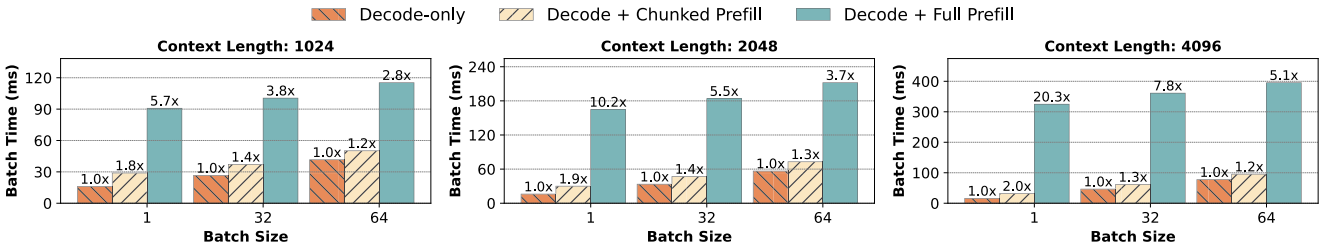
Takeaway-4: *There can be a large variance in compute time of LLM iterations depending on composition of prefill- and decode-tokens in the batch. This can lead to significant bubbles when using pipeline-parallelism.*

4 Sarathi-Serve: Design and Implementation

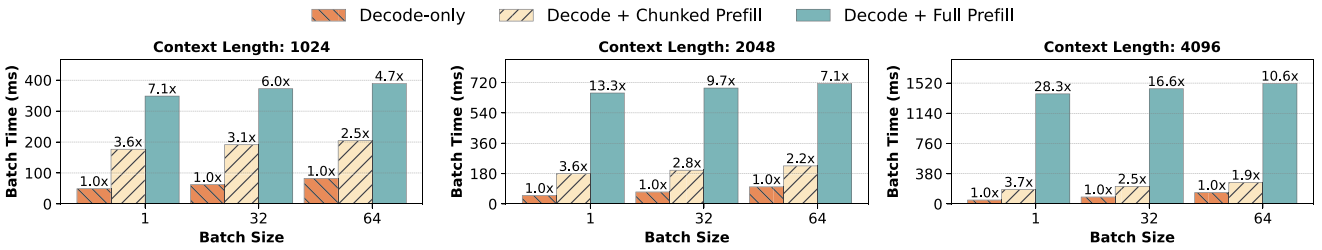
We now discuss the design and implementation of Sarathi-Serve — a system that provides high throughput with predictable tail latency via two key techniques – *chunked-prefills* and *stall-free batching*.

4.1 Chunked-prefills

As we show in §3.1, decode batches are heavily memory bound with low arithmetic intensity. This slack in arithmetic intensity presents an opportunity to piggyback additional computation in decode batches. Naively, this can be done by creating hybrid batches which combine the memory bound decodes along with compute bound prefills. However, in many practical scenarios, input prompts contain several thousand tokens



(a) Mistral-7B on one A100s with token budget of 256.



(b) LLaMA2-70B on four A100s with token budget of 512.

Figure 9: The incremental cost of coalescing prefills with decode batches. We consider two batching schemes – (i) Decode + Full Prefill represents the hybrid batching of Orca wherein the entire prefill is executed in a single iteration along with ongoing decodes. (ii) Decode + Chunked Prefill represents Sarathi-Serve wherein prefills are chunked before being coalesced with ongoing decodes with a fixed token budget. Sarathi-Serve processes prefill tokens with much lower impact on the latency of decodes. Further, the relative impact of Sarathi-Serve on latency reduces with higher decode batch size and context lengths.

on average *e.g.,* Table 2 shows that the median prompt size in *openchat_sharegpt4* and *arxiv_summarization* datasets is 1730 and 7059 respectively. Combining these long prefills with decode iterations would lead to high TBT latency.

To tackle this challenge, we present a technique called *chunked-prefills* which allows computing large prefills in small chunks across several iterations. *Chunked-prefills* is a prefill splitting mechanism hinged on two key insights. First, as discussed in §3.1, a prefill request with modest sequence length can effectively saturate GPU compute. For example, in Figure 4, prefill throughput starts saturating around sequence length of 512 tokens. Second, in many practical scenarios, input prompts contain several thousand tokens on average (Table 2). This provides an opportunity to break large prefill requests into smaller units of compute which are still large enough to saturate GPU compute. In Sarathi-Serve, we leverage this mechanism to form batches with appropriate number of tokens such that we can utilize the compute potential in decode batches without violating the TBT SLO.

4.2 *Stall-free batching*

The Sarathi-Serve scheduler is an iteration-level scheduler that leverages *chunked-prefills* and coalescing of prefills and decodes to improve throughput while minimizing latency.

Unlike Orca and vLLM which stall existing decodes to execute prefills, Sarathi-Serve leverages the arithmetic intensity slack in decode iterations to execute prefills without delaying the execution of decode requests in the system. We call this approach *stall-free batching* (Algorithm 3). Sarathi-Serve first calculates the budget of maximum number of tokens that can be executed in a batch based on user specified SLO. We describe the considerations involved in determining this token budget in depth in §4.3. In every scheduling iteration, we first pack all the running decodes in the next batch (lines 6-8 in Algorithm 3). After that, we include any partially completed prefill (lines 9-12). Only after all the running requests have been accommodated, we admit new requests (lines 13-20). When adding prefill requests to the batch, we compute the maximum chunk size that can be accommodated within the leftover token budget for that batch (lines 11, 15). By restricting the computational load in every iteration, *stall-free batching* ensures that decodes never experience a generation stall due to a co-running prefill chunk. We compare the latency for hybrid batches with and without chunked prefills in Figure 9. Naive hybrid batching leads to dramatic increase of up to 28.3× in the TBT latency compared to a decode-only batch. In contrast, Sarathi-Serve provides a much tighter bound on latency with chunking.

Figure 7 shows the scheduling policy of Sarathi-Serve in action, for the same example used in §3.2. The first iteration is decode-only as there are no prefills to be computed. However, after a new request C enters the system, Sarathi-Serve first splits the prefill of C into two chunks and schedules them in

Algorithm 3 *Stall-free batching* with Sarathi-Serve. First the batch is filled with with ongoing decode tokens (lines 6-8) and optionally one prefill chunk from ongoing (lines 10-12). Finally, new requests are added (lines 13-20) within the token budget so as to maximize throughput with minimal latency impact on the TBT of delaying the ongoing decodes.

```
1:  Input: T_max, Application TBT SLO.
2:  Initialize token_budget, τ ← compute_token_buget(T_max)
3:  Initialize batch_num_tokens, n_t ← 0
4:  Initialize current batch B ← ∅
5:  while True do
6:      for R in B do
7:          if is_prefill_complete(R) then
8:              n_t ← n_t + 1
9:      for R in B do
10:         if not is_prefill_complete(R) then
11:             c ← get_next_chunk_size(R, τ, n_t)
12:             n_t ← n_t + c
13:     R_new ← get_next_request()
14:     while can_allocate_request(R_new) ∧ n_t < τ do
15:         c ← get_next_chunk_size(R_new, τ, n_t)
16:         if c > 0 then
17:             n_t ← n_t + c
18:             B ← R_new
19:         else
20:             break
21:
22:     process_hybrid_batch(B)
23:     B ← filter_finished_requests(B)
24:     n_t ← 0
```

subsequent iterations. At the same time, with *stall-free batching*, it coalesces the chunked prefills with ongoing decodes of A and B. This way, Sarathi-Serve stalls neither decodes nor prefills unlike existing systems, allowing Sarathi-Serve to be largely free of latency spikes in TBT without compromising throughput. Furthermore, *stall-free batching* combined with *chunked-prefills* also ensures uniform compute hybrid batches in most cases, which helps reduce bubbles when using pipeline parallelism, thereby enabling efficient and scalable deployments.

4.3 Determining Token Budget

The token budget is determined based on two competing factors — TBT SLO requirement and *chunked-prefills* overhead. From a TBT minimization point of view, a smaller token budget is preferable because iterations with fewer prefill tokens have lower latency. However, smaller token budget can result in excessive chunking of prefills resulting in overheads due to 1) lower GPU utilization and 2) repeated KV-cache access in the attention operation which we discuss below.

During the computation of *chunked-prefills*, the attention operation for every chunk of a prompt needs to access the KV-cache of *all* prior chunks of the same prompt. This results in increased memory reads from the GPU HBM even though the computational cost is unchanged. For example, if a prefill sequence is split into N chunks, then the first chunk's KV-cache is loaded $N-1$ times, the second chunk's KV-cache is loaded $N-2$ times, and so on. However, we find that even at small chunk sizes attention prefill operation is compute bound operation. In practice, there can be small overhead associated with chunking due to fixed overheads of kernel launch, etc. We present a detailed study of the overheads of *chunked-prefills* in §5.4.

Thus, one needs to take into account the trade-offs between prefill overhead and decode latency while determining the token budget. This can be handled with a one-time profiling of batches with different number of tokens and setting the token budget to maximum number of tokens that can be packed in a batch without violating TBT SLO.

Another factor that influences the choice of token budget is the *tile-quantization* effect [13]. GPUs compute matmuls by partitioning the given matrices into tiles and assigning them to different thread blocks for parallel computation. Here, each thread block refers to a group of GPU threads and computes the same number of arithmetic operations. Therefore, matmuls achieve maximum GPU utilization when the matrix dimensions are divisible by the tile size. Otherwise, due to *tile-quantization*, some thread blocks perform extraneous computation [13]. We observe that tile-quantization can dramatically increase prefill computation time *e.g.,* in some cases, using chunk size of 257 can increase prefill time by 32% compared to that with chunk size 256.

Finally, when using pipeline parallelism the effect of token budget on pipeline bubbles should also be taken into account. Larger chunks lead to higher inter-batch runtime variations that result in pipeline bubbles which results in lower overall system throughput. On the other hand, picking a very small token budget can lead to higher overhead due to lower arithmetic intensity and other fixed overheads.

Therefore, selecting a suitable token budget is a complex decision which depends on the desired TBT SLO, parallelism configuration, and specific hardware properties. We leverage Vidur [28], a LLM inference profiler and simulator to determine the token budget that maximizes system capacity under specific deployment scenario.

4.4 Implementation

We implement Sarathi-Serve on top of the open-source implementation of vLLM [23, 53]. We added support for paged chunk prefill using FlashAttention v2 [38] and FlashInfer [74] kernels. We use FlashAttention backend for all the evaluations in this paper due to its support for wider set of models. We also extend the base vLLM codebase to support various

Model	Attention Mechanism	GPU Configuration	Memory Total (per-GPU)
Mistral-7B	GQA-SW	1 A100	80GB (80GB)
Yi-34B	GQA	2 A100s (TP2)	160GB (80GB)
LLaMA2-70B	GQA	8 A40s (TP4-PP2)	384GB (48GB)
Falcon-180B	GQA	4 A100s×2 nodes (TP4-PP2)	640GB (80GB)

Table 1: Models and GPU configurations (GQA: grouped-query attention, SW: sliding window).

Dataset	Prompt Tokens			Output Tokens		
	Median	P90	Std.	Median	P90	Std.
openchat_sharegpt4	1730	5696	2088	415	834	101
arxiv_summarization	7059	12985	3638	208	371	265

Table 2: Datasets used for evaluation.

scheduling policies, chunked prefills, pipeline parallelism and an extensive telemetry system. We use NCCL [15] for both pipeline and tensor parallel communication. Source code for the project is available at https://github.com/microsoft/sarathi-serve.

5 Evaluation

We evaluate Sarathi-Serve on a variety of popular models and GPU configurations (see Table 1) and two datasets (see Table 2). We consider vLLM and Orca as baseline because they represent the state-of-the-art for LLM inference. Our evaluation seeks to answer the following questions:

1. What is the maximum load a model replica can serve under specific Service Level Objective (SLO) constraints with different inference serving systems (§5.1) and how does this load vary with varying SLO constraints (§5.2)?
2. How does Sarathi-Serve perform under various deployments such as TP and PP? (§5.3)
3. What is the overhead of *chunked-prefills*? (§5.4.1)
4. What is the effect of each of *chunked-prefills* and *stall-free batching* in isolation as opposed to using them in tandem? (§5.4.2)

Models and Environment: We evaluate Sarathi-Serve across four different models Mistral-7B [51], Yi-34B [24], LLaMA2-70B [66] and Falcon-180B [31] – these models are among the best in their model size categories. We use two different server configurations. For all models except LLaMA2-70B we use Azure NC96ads v4 VMs, each equipped with 4 NVIDIA 80GB A100 GPUs, connected with pairwise NVLINK. The machines are connected with a 100 Gbps ethernet connection. For LLaMA2-70B, we use a server with eight pairwise connected NVIDIA 48GB A40 GPUs. We run Yi-34B in a 2-way tensor parallel configuration (TP-2), and LLaMA2-70B and Falcon-180B in a hybrid parallel configuration with four tensor parallel workers and two pipeline stages for (TP4-PP2).

Workloads: In order to emulate the real-world serving scenarios, we generate traces by using the request length characteristics from the *openchat_sharegpt4* [68] and *arxiv_summarization* [36] datasets (Table 2). The *openchat_sharegpt4* trace contains user-shared conversations with ChatGPT-4 [6]. A conversation may contain multiple rounds of interactions between the user and chatbot. Each such interaction round is performed as a separate request to the system. This multi-round nature leads to high relative variance in the prompt lengths. In contrast, *arxiv_summarization* is a collection of scientific publications and their summaries (abstracts) on arXiv.org [3]. This dataset contains longer prompts and lower variance in the number of output tokens, and is representative of LLM workloads such as Microsoft M365 Copilot [14] and Google Duet AI [10] etc. The request arrival times are generated using Poisson distribution. We filter outliers of these datasets by removing requests with total length more than 8192 and 16384 tokens, respectively.

Metrics: We focus on the median value for the TTFT since this metric is obtained only once per user request and on the 99th percentile (P99) for TBT values since every decode token results in a TBT latency value.

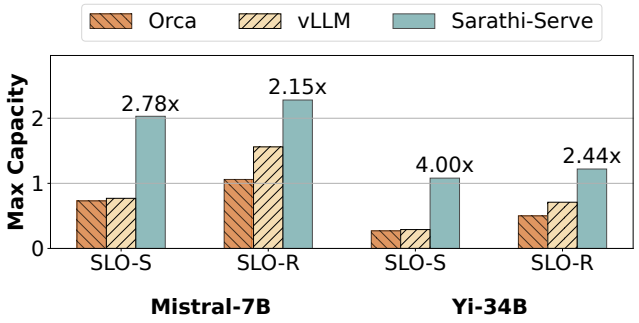
Model	*relaxed* SLO P99 TBT (s)	*strict* SLO P99 TBT (s)
Mistral-7B	0.5	0.1
Yi-34B	1	0.2
LLaMA2-70B	5	1
Falcon-180B	5	1

Table 3: SLOs for different model configurations.

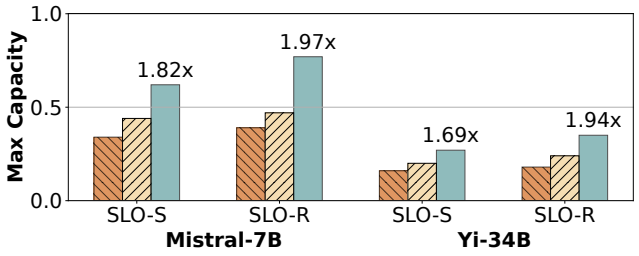
5.1 Capacity Evaluation

We evaluate Sarathi-Serve, Orca and vLLM on all four models and both datasets under two different latency configurations: *relaxed* and *strict*. Similar to Patel et al. [58], to account for the intrinsic performance limitations of a model and hardware pair, we define the SLO on P99 TBT to be equal to 5× and 25× the execution time of a decode iteration for a request (with prefill length of 4k and 32 batch size) running without any prefill interference for the *strict* and *relaxed* settings, respectively. Table 3 shows a summary of the absolute SLO thresholds. Note that the *strict* SLO represents the latency target desired for interactive applications like chatbots. On the other hand, the *relaxed* configuration is exemplary of systems where the complete sequence of output tokens should be generated within a predictable time limit but the TBT constraints on individual tokens is not very strict. For all load experiments, we ensure that the maximum load is sustainable, i.e., the queuing delay does not blow up (we use a limit of 2 seconds on median scheduling delay).

Figure 10 and Figure 11 show the results of our capacity

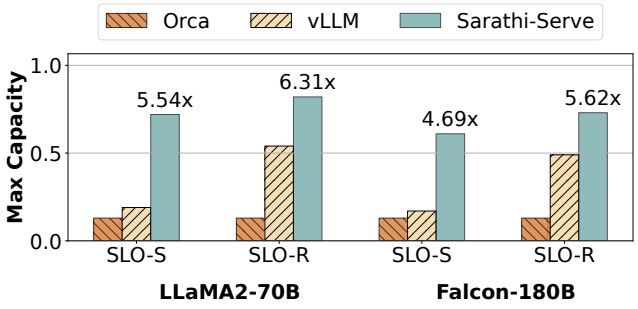


(a) Dataset: *openchat_sharegpt4*.

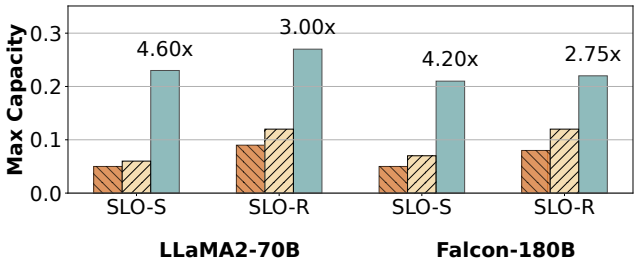


(b) Dataset: *arxiv_summarization*.

Figure 10: Capacity (in queries per second) of Mistral-7B and Yi-34B with different schedulers under strict (SLO-S) and relaxed (SLO-R) latency SLOs.



(a) Dataset: *openchat_sharegpt4*.



(b) Dataset: *arxiv_summarization*.

Figure 11: Capacity of LLaMA2-70B and Falcon-180B (models with pipeline parallelism) with different schedulers under strict (SLO-S) and relaxed (SLO-R) latency SLOs.

experiments. Sarathi-Serve consistently outperforms Orca and vLLM in all cases across models and workloads. Under *strict* SLO, Sarathi-Serve can sustain up to 4.0× higher load compared to Orca and 3.7× higher load than vLLM under *strict* SLO (Yi-34B, *openchat_sharegpt4*). For larger models using pipeline parallelism, Sarathi-Serve achieves gains of up to 6.3× and 4.3× compared to Orca and vLLM respectively (LLaMA2-70B, *openchat_sharegpt4*) due to few pipeline bubbles.

We observe that in most scenarios, Orca and vLLM violate the P99 TBT latency SLO before they can reach their maximum serviceable throughput. Thus, we observe relaxing the latency target leads to considerable increase in their model serving capacity. In Sarathi-Serve, one can adjust the chunk size based on the desired SLO. Therefore, we use a strict token budget and split prompts into smaller chunks when operating under *strict* latency SLO. This reduces system efficiency marginally but allows us to achieve lower tail latency. On the other hand, when the latency constraint is relaxed, we increase the token budget to allow more efficient prefills. We use token budget of 2048 and 512 for all models under the *relaxed* and *strict* settings, respectively, except for the LLaMA2-70B *relaxed* configuration where we use token budget of 1536 to reduce the impact of pipeline bubbles. The system performance can be further enhanced by dynamically varying the token budget based on workload characteristics. We leave this exploration for future work.

We further notice that vLLM significantly outperforms Orca under relaxed setting. The reason for this is two-fold. First, Orca batches prompts for multiple requests together (*max sequence length * batch size* compared to *max sequence length* in vLLM), which can lead to even higher tail latency in some cases. Second, vLLM supports a much larger batch size compared to Orca. The lower batch size in Orca is due to the lack of PagedAttention and the large activation memory footprint associated with processing batches with excessively large number of tokens.

Finally, note that the capacity of each system is higher for *openchat_sharegpt4* dataset compared to the *arxiv_summarization* dataset. This is expected because prompts in the *arxiv_summarization* datasets are much longer - 7059 vs 1730 median tokens as shown in Table 2. The larger prompts makes Orca and vLLM more susceptible to latency violations due to higher processing time of these longer prefills.

5.2 Throughput-Latency Tradeoff

To fully understand the throughput-latency tradeoff in LLM serving systems, we vary the P99 TBT latency SLO and observe the impact on system capacity for vLLM and Sarathi-Serve. Figure 12 shows the results for Mistral-7B and Yi-34B models with five different SLO values for the *openchat_sharegpt4* dataset.

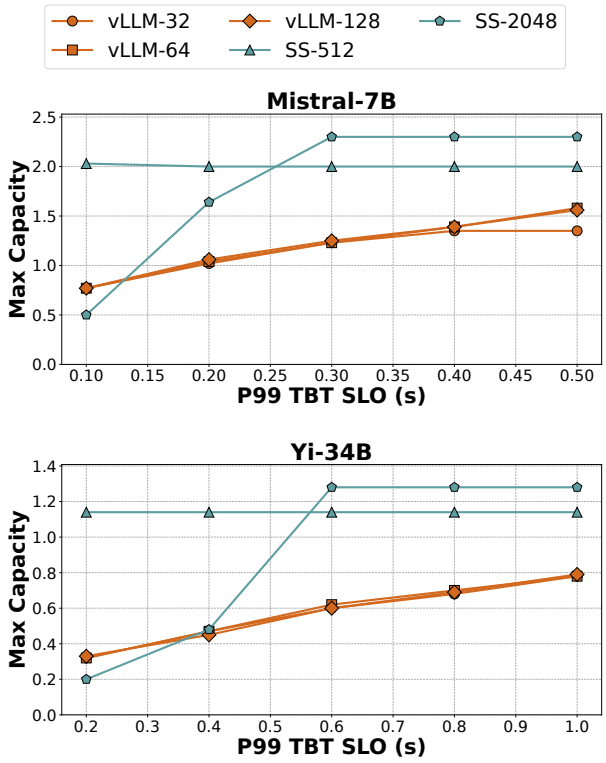
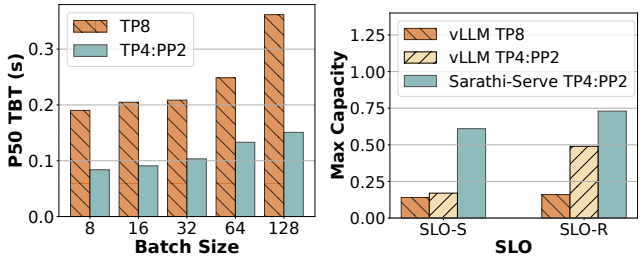


Figure 12: Latency – Throughput tradeoff in vLLM and Sarathi-Serve for Mistral-7B and Yi-34B models on *openchat_sharegpt4* dataset. We evaluate vLLM with three different max batch sizes of 32, 64 and 128. For Sarathi-Serve, we consider token budget of 512 and 2048 with max batch size of 128. Sarathi-Serve delivers 3.5× higher capacity under stringent SLOs for Yi-34B using *Stall-free batching*.

We evaluate vLLM with three different batch sizes in an attempt to navigate the latency-throughput trade-off as prescribed by Yu et al. [75]. The maximum capacity of vLLM gets capped due to generation stalls under stringent TBT SLOs. Notably, the capacity of vLLM remains largely identical for all the three batch size settings. This implies that even though PagedAttention enables large batch sizes with efficient memory management – in practical situations with latency constraints, vLLM cannot leverage the large batch size due to the steep latency-throughput tradeoff made by it's *prefill-prioritizing* scheduler.

On the other hand, the latency-throughput tradeoff in Sarathi-Serve can be precisely controlled by varying the token budget. Sarathi-Serve achieves 3.5× higher capacity compared to vLLM under strict SLO (100ms, Mistral-7B) using a small token budget of 512. For scenarios with more relaxed SLO constraints, picking a larger token budget of 2048 allows Sarathi-Serve to operate more efficiently resulting in 1.65× higher capacity compared to vLLM (1s, Yi-34B).



(a) TBT (Falcon-180B). (b) Capacity (Falcon-180B).

Figure 13: TP scales poorly across nodes. (a) Median TBT for decode-only batches: cross node TP increases median TBT by more than 2× compared to a 4-way TP within node and PP across nodes. (b) Capacity under strict (SLO-S) and relaxed (SLO-R) latency SLOs: Sarathi-Serve increases Falcon-180B's serving capacity by 4.3× and 3.6× over vLLM's TP-only and hybrid-parallel configurations under strict SLOs.

5.3 Making Pipeline Parallel Viable

We now show that Sarathi-Serve makes it feasible to efficiently serve LLM inference across commodity networks with efficient pipeline parallelism. For these experiments, we run Falcon-180B over two nodes, each with four A100 GPUs, connected over 100 Gbps Ethernet. We evaluate model capacity under three configurations: vLLM with 8-way TP, vLLM with our pipeline-parallel implementation and Sarathi-Serve with pipeline-parallel. For PP configurations, we do 4-way TP within node and 2-way PP across nodes.

Figure 13a shows the latency for decode-only batches for Falcon-180B with purely tensor parallel TP-8 deployment compared to a TP-4 PP-2 hybrid parallel configuration. We observe that the median latency for tensor parallelism is $\sim 2\times$ higher than pipeline parallelism. This is because TP incurs high communication overhead due to cross-node all-reduces.

Figure 13b shows the capacity for tensor and hybrid parallel configurations for Falcon-180B on *openchat_sharegpt4* dataset. Note that unlike the hybrid parallel configuration, TP achieves low capacity even under the *relaxed* SLO due to high latency. Even though vLLM can support a fairly high load with hybrid parallelism under *relaxed* SLO, it's performance drops sharply under the *strict* regime due to pipeline bubbles. Sarathi-Serve on the other hand, leverages *chunked-prefills* to reduce the variation in the execution time between microbatches to avoid pipeline bubbles, resulting in a 1.48× increase in capacity under *relaxed* SLOs and 3.6× increase in capacity under *strict* SLOs.

5.4 Ablation Study

In this subsection, we conduct an ablation study on different aspects on Sarathi-Serve. In particular, we are interested in answering the following two questions: 1) what is the effect of

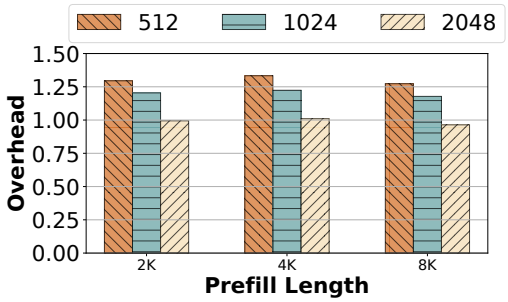


Figure 14: Overhead of *chunked-prefills* in prefill computation for Yi-34B (TP-2) normalized to the cost of no-chunking, shown for various prompt lengths using chunk lengths of 512, 1024 and 2048.

Scheduler	*openchat_sharegpt4*		*arxiv_summarization*	
	P50 TTFT	P99 TBT	P50 TTFT	P99 TBT
hybrid-batching-only	0.53	0.68	3.78	1.38
chunked-prefills-only	1.04	0.17	5.38	0.20
Sarathi-Serve (combined)	0.76	0.14	3.90	0.17

Table 4: TTFT and TBT latency measured in seconds for *hybrid-batching* and *chunked-prefills* used in isolation as well as when they are used in tandem, evaluated over 128 requests for Yi-34B running on two A100s with a token budget of 1024. By using both *hybrid-batching* and *chunked-prefills*, Sarathi-Serve is able to lower both TTFT and TBT.

chunking on prefill throughput, and 2) analyzing the impact of hybrid-batching and chunking on latency. While we provide results only for a few experiments in this section, all the trends discussed below are consistent across various model-hardware combinations.

5.4.1 Overhead of *chunked-prefills*

Figure 14 shows how much overhead chunking adds in Yi-34B – on overall prefill runtime. As expected, smaller chunks introduce higher overhead as shown by the gradually decreasing bar heights in Figure 14. However, even with the smallest chunk size of 512, we observe a moderate overhead of at most ∼25%. Whereas with the larger token budget of 2048, chunked prefills have almost negligible overhead.

5.4.2 Impact of individual techniques

Finally, Table 4 shows the TTFT and TBT latency with each component of Sarathi-Serve evaluated in isolation *i.e.,* *chunked-prefills*-only, *hybrid-batching*-only (mixed batches with both prefill and decode requests) and when they are used in tandem. These results show that the two techniques work best together: *chunked-prefills*-only increases TTFT as prefill chunks are slightly inefficient whereas *hybrid-batching*-only increases TBT because long prefills can still create generation stalls. When used together, Sarathi-Serve improves performance along both dimensions.

6 Related Work

Model serving systems: Systems such as Clipper [37], TensorFlow-Serving [56], Clockwork [45] and BatchMaker [44] study various placement, caching and batching strategies for model serving. However, these systems fail to address the challenges of auto-regressive transformer inference. More recently, systems such as Orca [75], vLLM [53], FlexGen [63], FasterTransformers [7], LightSeq [70], and TurboTransformers [42] propose domain-specific optimizations for transformer inference. FlexGen [63] optimizes LLM inference for throughput in resource-constrained offline scenarios i.e., it is not suitable for online serving. FastServe [72] proposed a preemptive scheduling framework for LLM inference to minimize the job completion times. We present a detailed comparison with Orca and vLLM as they represent the state-of-the-art in LLM inference.

Another approach that has emerged recently is to disaggregate the prefill and decode phases on different replicas as proposed in SplitWise, DistServe and TetriInfer [47, 58, 77]. These solutions can entirely eliminate the interference between prefills and decodes. However, disaggregation requires migrating the KV cache of *each* request upon the completion of its prefill phase which could be challenging in the absence of high-bandwidth interconnects between different replicas. In addition, this approach also under-utilizes the GPU memory capacity of the prefill replicas i.e., only the decode replicas are responsible for storing the KV cache. On the positive side, disaggregated approaches can execute prefills with maximum efficiency (and therefore yield better TTFT) unlike chunked prefills that are somewhat slower than full prefills. We leave a quantitative comparison between Sarathi-Serve and disaggregation-based solutions for future work.

Recently, Sheng et al. [62] proposed modification to iteration-level batching algorithm to ensure fairness among clients in a multi-tenant environment. FastServe [72] uses a preemption based scheduling mechanism to mitigate head-of-the-line blocking. Such algorithmic optimizations are complimentary to our approach and can benefit from lower prefill-decode interference enabled by Sarathi-Serve. Another recent system, APIServe [26] adopted chunked prefills from Sarathi to utilize wasted compute in decode batches for ahead-of-time prefill recomputation for multi-turn API serving.

Improving GPU utilization for transformers: Recent works have proposed various optimizations to improve the hardware utilization for transformers. FasterTransformer uses model-specific GPU kernel implementations. CocoNet [50] and [69] aim to overlap compute with communication to improve GPU utilization: these techniques are specially useful while using a high degree of tensor-parallel for distributed models

where communication time can dominate compute. Further, the cost of computing self-attention grows quadratically with sequence length and hence can become significant for long contexts. [38,39,60] have proposed various techniques to minimize the memory bottlenecks of self-attention with careful tiling and work partitioning. In addition, various parallelization strategies have been explore to optimize model placement. These techniques are orthogonal to Sarathi-Serve.

Model optimizations: A significant body of work around model innovations has attempted to address the shortcomings of transformer-based language models or to take the next leap forward in model architectures, beyond transformers. For example, multi-query attention [61] shares the same keys and values across all the attention heads to reduce the size of the KV-cache, allowing to fit a larger batch size on the GPUs. Several recent works have also shown that the model sizes can be compressed significantly using quantization [40, 41, 43, 73]. Mixture-of-expert models are aimed primarily at reducing the number of model parameters that get activated in an iteration [32, 48, 54]. More recently, retentive networks have been proposed as a successor to transformers [65]. In contrast, we focus on addressing the performance issues of popular transformer models from a GPU's perspective.

7 Conclusion

Optimizing LLM inference for high throughput and low latency is desirable but challenging. We presented a broad characterization of existing LLM inference schedulers by dividing them into two categories – *prefill-prioritizing* and *decode-prioritizing*. In general, we argue that the former category is better at optimizing throughput whereas the latter is better at optimizing TBT latency. However, none of them is ideal when optimizing throughput and latency are both important.

To address this tradeoff, we introduce Sarathi-Serve—a system that instantiates a novel approach comprised of *chunked-prefills* and *stall-free batching*. Sarathi-Serve chunks input prompts into smaller units of work to create stall-free schedules. This way, Sarathi-Serve can add new requests in a running batch without pausing ongoing decodes. Our evaluation shows that Sarathi-Serve improves the serving capacity of Mistral-7B by up to 2.6× on a single A100 GPU and up to 5.6× for Falcon-180B on 8 A100 GPUs.

8 Acknowledgement

We would like to thank OSDI reviewers and our shepherd for their insightful feedback. This research is partly supported by GT Cloud Hub, under the auspices of the Institute for Data Engineering and Science (IDEaS), with funding from Microsoft, and the Center for Research into Novel Compute Hierarchies (CRNCH) at Georgia Tech.

References

[1] Amazon codewhisperer. https://aws.amazon.com/codewhisperer/.

[2] Anthropic claude. https://claude.ai.

[3] arxiv.org e-print archive. https://arxiv.org/.

[4] Bing ai. https://www.bing.com/chat.

[5] Character ai. https://character.ai.

[6] Chatgpt. https://chat.openai.com.

[7] Faster Transformer. https://github.com/NVIDIA/FasterTransformer.

[8] Github copilot. https://github.com/features/copilot.

[9] Google bard. https://bard.google.com.

[10] Google duet ai. https://workspace.google.com/solutions/ai/.

[11] Komo. https://komo.ai/.

[12] Lightllm: A light and fast inference service for llm. https://github.com/ModelTC/lightllm.

[13] Matrix multiplication background user's guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[14] Microsoft copilot. https://www.microsoft.com/en-us/microsoft-copilot.

[15] Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl.

[16] Nvidia dgx platform. https://www.nvidia.com/en-us/data-center/dgx-platform/.

[17] NVIDIA Triton Dynamic Batching. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_configuration.html#dynamic-batcher.

[18] Openai gpt-3: Understanding the architecture. https://www.theaidream.com/post/openai-gpt-3-understanding-the-architecture.

[19] Perplexity ai. https://www.perplexity.ai/.

[20] Replit ghostwriter. https://replit.com/site/ghostwriter.

[21] Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM.

[22] Using NVIDIA's AI/ML Frameworks for Generative AI on VMware vSphere. https://core.vmware.com/blog/using-nvidias-aiml-frameworks-generative-ai-vmware-vsphere.

[23] vllm: Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm.

[24] Yi series of large language models trained from scratch by developers at 01.AI. https://huggingface.co/01-ai/Yi-34B-200K.

[25] You.com. https://you.com/.

[26] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. Apiserve: Efficient api support for large-language model inferencing. *arXiv preprint arXiv:2402.01869*, 2024.

[27] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.

[28] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference. *Proceedings of The Seventh Annual Conference on Machine Learning and Systems, 2024, Santa Clara*, 2024.

[29] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

[30] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[31] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models, 2023.

[32] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient large scale language modeling with mixtures of experts, 2022.

[33] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[34] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[35] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

[36] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 615–621, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[37] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[38] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

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

[40] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[41] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[42] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient GPU serving system for transformer models. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 389–402. ACM, 2021.

[43] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[44] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[46] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

[47] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[48] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Hsien-Hsin S. Lee, Anjali Sridhar, Shruti Bhosale, Carole-Jean Wu, and Benjamin Lee. Towards moe deployment: Mitigating inefficiencies in mixture-of-expert (moe) inference, 2023.

[49] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[50] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 402–416, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[52] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[53] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed MoE training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, Boston, MA, July 2023. USENIX Association.

[55] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[56] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving, 2017.

[57] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[58] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[59] 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.

[60] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

[61] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[62] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. *arXiv preprint arXiv:2401.00588*, 2023.

[63] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[64] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[65] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023.

[66] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[68] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. Openchat: Advancing open-source language models with mixed-quality data, 2023.

[69] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[70] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers (NAACL-HLT)*, pages 113–120. Association for Computational Linguistics, June 2021.

[71] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[72] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[73] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2023.

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

[75] 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.

[76] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

[77] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

A Artifact Appendix

Abstract

Our open source artifact is available on GitHub. This repository contains our implementation of Sarathi-Serve as well as the harnesses and scripts for running and plotting the experiments described in this paper.

This repository originally started as a fork of the vLLM project. Sarathi-Serve is a lightweight high-performance research prototype and doesn't have complete feature parity with open-source vLLM. We have only retained the most critical features and adopted the codebase for faster research iterations.

Scope

This artifact allows the readers to validate the claims made in the Sarathi-Serve paper (the figures) and provides a means to replicate the experiments described. The artifact can be used to set up the necessary environment, execute the main results, and perform microbenchmarks, thus providing a comprehensive understanding of the key claims in Sarathi-Serve.

Contents

The repository is structured as follows, the primary source code for the system is contained in directory */sarathi*. The implementations for custom CUDA kernels are within the */csrc* directory. All the scripts to reproduce the experiments are in */osdi-experiments* and finally, the trace files used for the experiments are stored in */data*.

Hosting

You can obtain our artifacts from GitHub: GitHub. The main branch of the Github repository is actively updated, but we will maintain clear and accessible instructions about our artifacts in an easily identifiable README file. All the detailed instructions and README files to reproduce the experiments in the OSDI paper are available in the branch *osdi-sarathi-serve*.

Requirements

Sarathi-Serve has been tested with CUDA 12.1 on A100 and A40 GPUs. The specific GPU SKUs on which the experiments were performed and the parallelism strategies used are clearly explained in the README corresponding to the figures in the artifact, for ease of reproducibility.