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

Stall 的意思是第 i 个 decoded token 到第 i+1 个 decoded token 的时间

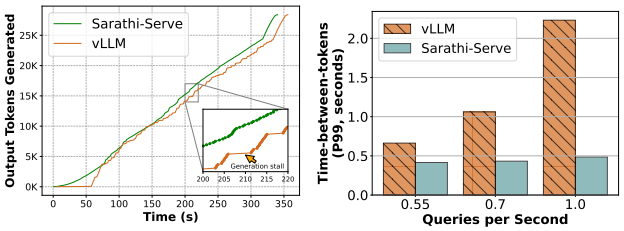
Abstract

Each LLM serving request goes through two phases. The first is *prefill* which processes the entire input prompt to produce one 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 inspired by the techniques we originally proposed for optimizing throughput in Sarathi [27]. Sarathi-Serve leverages *chunked-prefills* from Sarathi to create *stall-free* schedules that can add 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. Our evaluation shows that Sarathi-Serve improves serving throughput within desired latency SLOs of Mistral-7B by up to 2.6× on a single A100 GPU and up to 6.9× for Falcon-180B on 8 A100 GPUs over Orca and vLLM.

1 Introduction

Large language models (LLMs) [31,32,49,53,68] 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, 53], search [4, 9, 11, 18, 24], code assistants [1, 8, 19], 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 [27, 50, 55, 56, 60, 71, 73].



(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 [50]. 1b shows the impact of increasing load on tail latency. Sarathi-Serve serves LLM inference with low tail latency and without generation stalls.

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 are subjected to a severe 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 significant compromise on either throughput or latency e.g., Figure 1b shows that increasing request load significantly increases tail latency in a state-of-the-art LLM serving system vLLM [50].

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

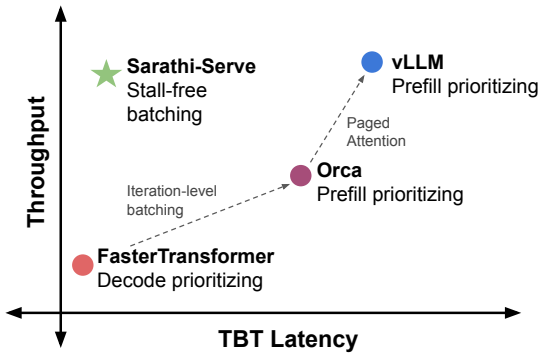


Figure 2: Current LLM serving systems involve a tradeoff between throughput and latency depending on their scheduling policy. Prioritizing prefills optimizes throughput but sacrifices TBT (time-between-tokens) tail latency whereas prioritizing decodes has the opposite effect. Sarathi-Serve serves high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

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 making 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 batch execution continues until the completion of the last request.

Orca [71] 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 wasteful computation of request-level batching systems. Orca and several other recent systems like vLLM [21] combine iteration-level batching with *prefill-prioritizing* scheduling wherein they preferentially 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 pauses ongoing decodes. Since prefills can take arbitrarily long time depending on the context length of given prompts, *prefill-prioritizing* schedulers lead to an undesirable phenomenon we refer to as *generation stalls* in this paper. For example, Figure 1a shows that generation stalls in vLLM can last over several seconds each.

We recently proposed Sarathi [27] – a throughput-oriented LLM inference engine based on two key ideas: *chunked-prefills* and *decode-maximal batching*. In Sarathi, *chunked-prefills* compute a prompt's prefill phase over multiple iterations (each with a subset of the prompt tokens) and *decode-maximal batching* coalesces on-going decodes with a prefill chunk. Coalescing enables computing decode tokens in a compute-bound regime whereas chunking prompts helps in maximizing the number of decode tokens that can be coalesced with prefills. Compute iterations in Sarathi are also uniform which improves distributed inference using pipeline-parallelism by minimizing pipeline bubbles. In [27], we demonstrated how these techniques can improve inference throughput. SplitFuse [44] is also based on these techniques.

Sarathi-Serve leverages Sarathi's mechanism and improves online inference with *stall-free* scheduling wherein *new requests join a running batch without pausing ongoing decodes*. Sarathi-Serve builds upon iteration-level batching but with an important distinction: it throttles the number of prefill tokens in each iteration 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 to throughput and TBT, time-to-first-byte (TTFT) is also an important performance metric for online LLM inference. TTFT denotes the latency between a request's arrival and when its first output token appears i.e., a request's TTFT is the sum of its scheduling delay and prefill computation time. Note that these metrics are not fully independent, i.e., increasing throughput also reduces TTFT. Sarathi-Serve enables low TBT latency and high throughput at the expense of a marginal increase in TTFT – this is due to the overhead of chunking. However, this trade-off between TBT and TTFT can be controlled by tuning the chunk size.

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 2.8× under different SLO targets. Similarly for Mistral-7B, we achieve up to 2.6× higher serving capacity. Sarathi-Serve also helps reduce the effect of pipeline bubbles on latency and throughput, resulting in up to 6.9× increase in end-to-end serving capacity for Falcon-180B.

[1]We classify recent schedulers Splitwise [55] and DistServe [73] under a third category "disaggregated" and discuss them in §6.

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 [17], LLaMA [63], Yi [23] 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 [64], 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 is 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 [26,43]. Finally, the second linear layer, transforms the token embedding back to the original dimension h.

2.2 LLM Inference Process

Autoregressive decoding: LLM inference request processing consists of two distinct phases – a *prefill* phase followed by a *decode* phase. The prefill phase processes the entire user 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, 61, 71].

A typical LLM prompt contains 100s-1000s of input tokens Table 2, [72]. 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.

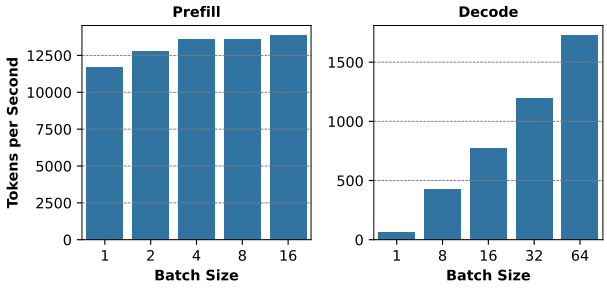


Figure 3: Throughput of the prefill and decode phases with different batch sizes for Mistral-7B running on a single A100 GPU. We use prompt length of 1024 for both prefill and decode experiments. *Batching boosts decode throughput almost linearly but has a marginal effect on prefill throughput.*

Batched LLM inference in multi-tenant environment: In production serving systems, the inference 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 inference 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. On the contrary, prefill phase of each request contains sufficient number of token to saturate GPU compute, which renders batching ineffective. Figure 3 illustrates throughput as a function of batch size, and we can observe that while for decode iterations throughput increases almost 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.*

Recently, several complementary techniques have been proposed to optimize throughput by enabling support for larger batch sizes. Kwon et al. propose PagedAttention [50], which allows more requests to concurrently execute, eliminating fragmentation in *KV-cache*. The use of Multi Query Attention (MQA) [58], Group Query Attention (GQA) [28] in leading edge LLM models like LLaMA2 [63], Falcon [29] and Yi [23] 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.

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. Prefills are executed eagerly (lines 8-9), potentially introducing a generations 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)
```

2.3 Performance Metrics

Most LLM applications, e.g. ChatGPT [6], Bing AI [4], etc. follow the streaming model for request response, where tokens are streamed to the user as they are generated. In this context, there are two primary latency metrics of interest: 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 latency.

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 inference.

2.4 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 [16] 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 [71].

To avoid wasted compute of request-level batching, Orca [71] 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 [21], TensorRT-LLM [20], and LightLLM [12] have adopted iteration-level batching.

Current iteration-level batching systems such as vLLM [21] and Orca [71] 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

3.1 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 as *generation stalls* as discussed below.

Figure 4 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 (lines 8-9 in Algorithm 2) 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 in subsequent decode iterations.

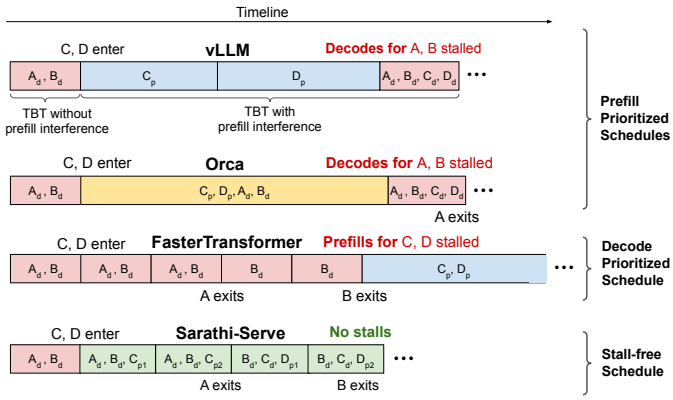


Figure 4: A generation stall occurs when one or more prefills are scheduled in between consecutive decode iterations of a request. A, B, C and D represent different requests. Subscript d represents a decode iteration, p represents a full prefill and $p0$, $p1$ represent two chunked prefills of a given 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 batch size in many decode iterations. In contrast, Sarathi-Serve generates a schedule that eliminates generation stalls yet delivers high throughput.

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 lengths of input prompts. Therefore, *prefill-prioritizing* schedulers can introduce *generation stalls* for ongoing decodes resulting in latency spikes due to 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 4, 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. [50] 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 [71]. However, lowering batch size adversely impacts throughput as shown in Figure 3 and therefore existing systems are forced to trade throughput or latency depending on the desired SLOs.

Takeaway-2: *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.*

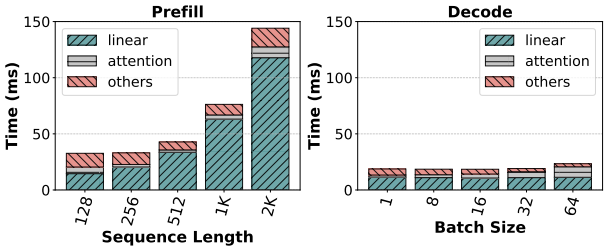


Figure 5: Prefill and decode time with different input sizes for Mistral-7B running on single A100 GPU. Linear layers contribute to the majority of runtime in both prefill and decode phases. Due to the low arithmetic intensity in decode batches, the cost of linear operation for 1 decode token is nearly same as 128 prefill tokens.

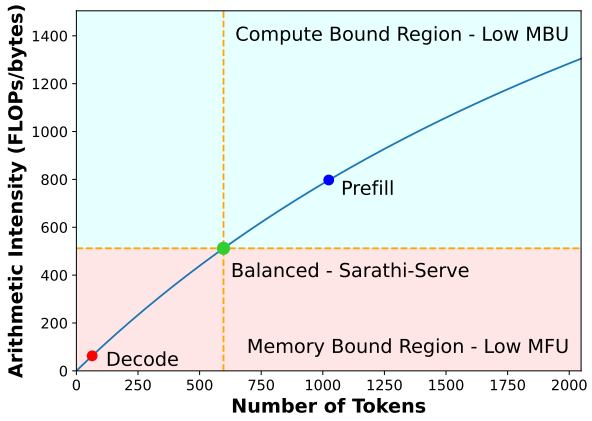


Figure 6: Arithmetic intensity trend for LLaMA2-70B linear operations with different number of token running on four A100s. Decode batches have low arithmetic intensity *i.e.,* they are bottlenecked by memory fetch time, leading to low compute utilization. Prefill batches are compute bound with sub-optimal bandwidth utilization. Sarathi forms hybrid batches by combining decodes and prefill chunks to maximize both compute and bandwidth utilization.

3.2 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. Figure 5 shows that the majority of the time in LLM inference is spent in linear operators. Therefore we focus our analysis on linear operators of the FFN module.

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 Model FLOPs Utilization

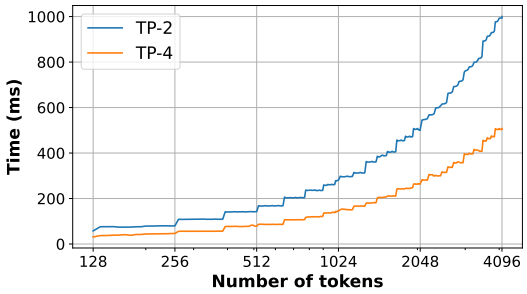


Figure 7: Linear layer execution time as function of number of tokens in a batch for LLaMA2-70B on A100(s) with different tensor parallel degrees. When the number of tokens is small, the the execution time is dictated by the cost of fetching weights from HBM memory. As a result, the execution time is 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.

(MFU) [32], 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 6 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 7 shows the total execution time of linear operators in 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-3: *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 the latency of the decode iteration.*

4 Sarathi-Serve: Design and Implementation

We now discuss the design and implementation of Sarathi-Serve which uses the techniques *chunked-prefills* defined in our prior work Sarathi [27] to create a *stall-free batching* scheduler optimized for online inference serving.

[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.

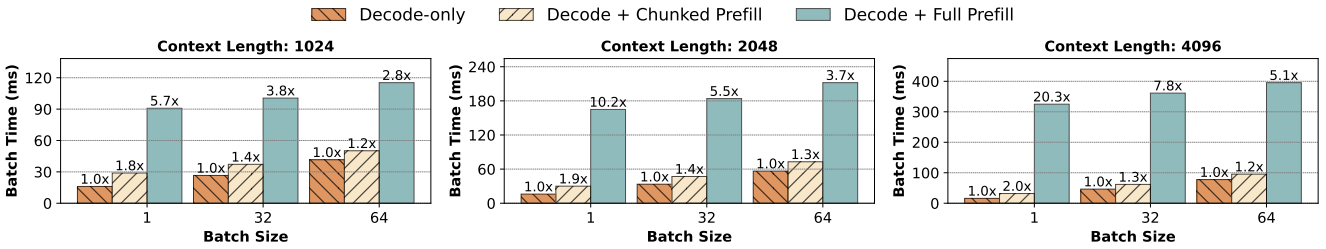
Algorithm 3 *Stall-free batching* with Sarathi-Serve. First the batch is filled with with ongoing decode tokens (lines 6-8) and optionally one prefill chunk from ongoing (lines 10-12). Finally, new requests are added (lines 13-20) within the token budget so as to maximize throughput with minimal latency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

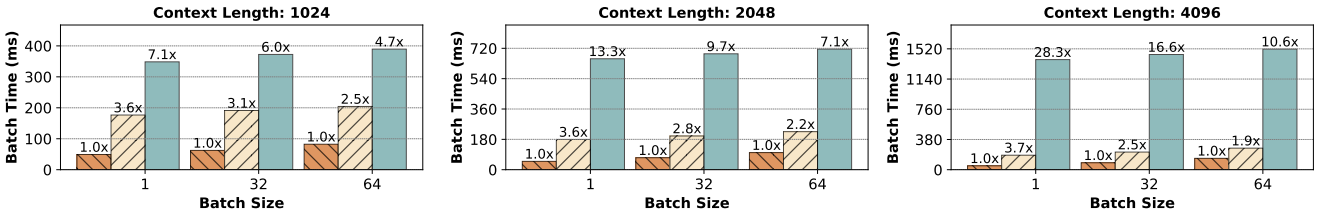
4.1 Chunked-prefills

As we show in §3.2, 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 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.

In our previous work Sarathi, we presented a technique called *chunked-prefills* which allows computing large prefills in small chunks across several iterations. 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.



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



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

Figure 8: 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.

4.2 *Stall-free batching*

The Sarathi-Serve scheduler is an iteration-level scheduler that leverages *chunked-prefills* and coalescing of prefills and decodes from Sarathi 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 generations stall due to a co-running prefill chunk. We compare the latency for hybrid batches with and without chunked prefills in Figure 8. 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 4 shows the scheduling policy of Sarathi-Serve in action, for the same example used in §3.1. 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 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.

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, the overhead due to increased attention time is not significant as attention computation is a relatively small fraction of the overall time. We present a detailed study of the overheads of *chunked-prefills* in §5.2.

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 (wasted) 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.

Therefore, selecting a suitable token budget is a two-fold decision. First, pick a token budget based on the desired TBT SLO. Next, ensure that the total number of tokens in a batch is a multiple of the tile size. The latter is relatively straightforward since the size of tile dimensions on GPUs is almost always a power of two.

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 various Service Level Objective (SLO) constraints with different inference serving systems?
2. What is the overhead of *chunked-prefills*?
3. What is the effect of each of *chunked-prefills* and *stall-free batching* in isolation as opposed to using them in tandem?

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.

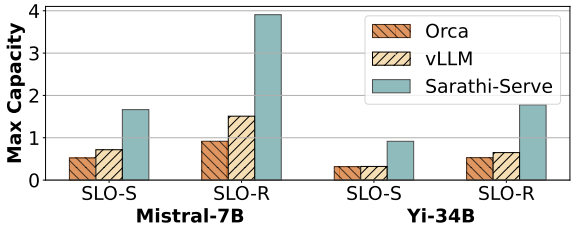
Implementation: We implement Sarathi-Serve on top of the open-source implementation of vLLM [21, 50]. The framework leverages PyTorch [54] and xFormers [22] for matrix multiplication and attention kernels respectively. We extend the base vLLM codebase to support various scheduling policies, chunked prefills, pipeline parallelism and an extensive telemetry system. We use NCCL [15] for both pipeline and tensor parallel communication.

Models and Environment: We evaluate Sarathi-Serve across four different models Mistral-7B [48], Yi-34B [23], LLaMA2-70B [63] and Falcon-180B [29] – 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).

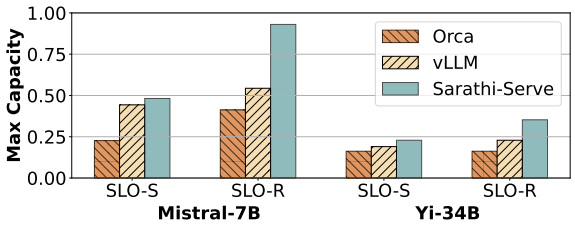
Workloads: In order to emulate the real-world serving scenarios, we generate traces by using the request length characteristics from the *openchat_sharegpt4* [65] and *arxiv_summarization* [33] datasets (Table 2). The *openchat_sharegpt4* trace contains user-shared conversations with ChatGPT-4 [6]. A 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

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.

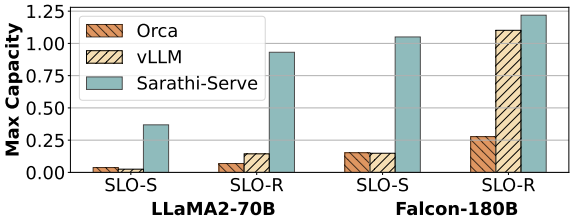


(a) Dataset: *openchat_sharegpt4*.

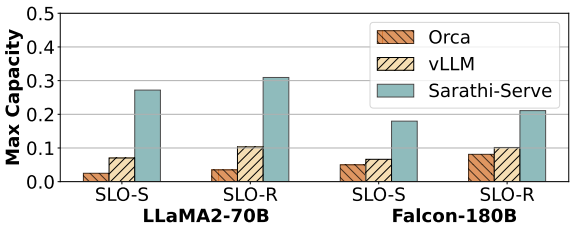


(b) Dataset: *arxiv_summarization*.

Figure 9: 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 10: Capacity of LLaMA2-70B and Falcon-180B (models with pipeline parallelism) with different schedulers under strict (SLO-S) and relaxed (SLO-R) latency SLOs.

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.

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. [55], 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 single request (with prefill length of 4k and 32 batch size) running without any contention for the *strict* and *relaxed* settings, respectively. Table 3 shows a summary of the absolute SLO thresholds. Note that the *strict* target represents the target SLOs 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 median scheduling delay).

Figure 9 and Figure 10 show the results of our capacity experiments. Sarathi-Serve consistently outperforms Orca and vLLM in all cases across models and workloads. Under *strict* SLO, Sarathi-Serve can sustain up to 3.2× higher load compared to Orca (Mistral-7B, *openchat_sharegpt4*) and 2.7× higher load than vLLM under *relaxed* SLO (Yi-34B, *openchat_sharegpt4*). The serving capacity of the bigger 70B and 180B models drop significantly compared to smaller models. However, even for these bigger models, Sarathi-Serve consistently outperforms Orca and vLLM.

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 a future work.

We further notice that vLLM significantly outperforms Orca. 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.

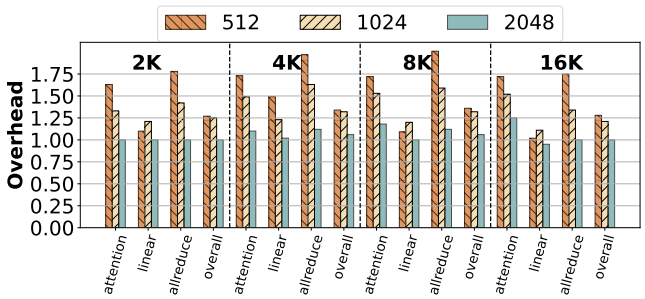


Figure 11: Overhead of *chunked-prefills* for Yi-34B (TP-2) shown for various operations and prompt sizes using batch size 1. To calculate the overhead, we divide prefill time with chunking by prefill time without chunking.

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.

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 higher prompts makes Orca and vLLM more susceptible to latency violations due to higher processing time of these longer prefills.

5.2 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 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.

Figure 11 shows how much overhead chunking adds in Yi-34B – on individual operators as well as the overall runtime. First, smaller chunks introduce higher overhead as shown by the gradually decreasing bar heights in Figure 11. This is expected and remains true in all cases, irrespective of the operator type or the prompt size. Second, attention and allreduce operators experience the highest overhead. The attention overhead is expected due to repeated global memory accesses. The overhead of allreduce is higher due to the fixed cost of NCCL communication collectives that get invoked more frequently with smaller chunks. Third, the linear operators experience the lowest overhead in all cases where the overhead is close to zero with chunk size 2048. Note that because the cost of linear operators dominate in transformers [71], the overall runtime overhead of *chunked-prefills* is close to that of the linear operators.

Scheduler	*openchat_sharegpt4*		*arxiv_summarization*	
	P50 TTFT	P99 TBT	P50 TTFT	P99 TBT
hybrid-batching-only	0.56	1.08	4.18	1.76
chunked-prefills-only	1.04	0.34	4.86	0.38
Sarathi-Serve (combined)	0.85	0.29	3.03	0.35

Table 4: Performance impact of *hybrid-batching* and *chunked-prefills* 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.

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 [34], TensorFlow-Serving [52], Clockwork [42] and BatchMaker [41] 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 [71], vLLM [50], FlexGen [60], FasterTransformers [7], LightSeq [67], and TurboTransformers [39] propose domain-specific optimizations for transformer inference. FlexGen [60] optimizes LLM inference for throughput in resource-constrained offline scenarios i.e., it is not suitable for online serving. FastServe [69] 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 [45, 55, 73]. 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. [59] proposed modification to iteration-level batching algorithm to ensure fairness among clients in a multi-tenant environment. FastServe [69] 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 [25] 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 [47] and [66] 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. [35,36,57] 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 [58] 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 [37, 38, 40, 70]. Mixture-of-expert models are aimed primarily at reducing the number of model parameters that get activated in an iteration [30, 46, 51]. More recently, retentive networks have been proposed as a successor to transformers [62]. 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.

We also introduce Sarathi-Serve and show that the techniques we originally proposed in Sarathi [27] (to optimize throughput alone) are equally powerful in co-optimizing throughput and latency. 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 6.9× for Falcon-180B on 8 A100 GPUs.

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

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

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

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

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

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

[22] XFORMERS OPTIMIZED OPERATORS. https://facebookresearch.github.io/xformers/components/ops.html.

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

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

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

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

[39] 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.

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

[41] 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.

[42] 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.

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

[44] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

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

[56] 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.

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

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

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

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

[66] 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.

[67] 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.

[68] 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.

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

[70] 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.

[71] 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.

[72] 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.

[73] 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.