SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills

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

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

Abstract

Large Language Model (LLM) inference consists of two distinct phases – *prefill* phase which processes the input prompt and *decode* phase which generates output tokens autoregressively. While the prefill phase effectively saturates GPU compute at small batch sizes, the decode phase results in low compute utilization as it generates one token at a time per request. The varying prefill and decode times also lead to imbalance across micro-batches when using pipeline-parallelism, resulting in further inefficiency due to bubbles.

We present SARATHI to address these challenges. SARATHI employs *chunked-prefills*, which splits a prefill request into equal sized chunks, and *decode-maximal batching*, which constructs a batch using a single prefill chunk and populates the remaining slots with decodes. During inference, the prefill chunk saturates GPU compute, while the decode requests 'piggyback' and cost up to an order of magnitude less compared to a decode-only batch. *Chunked-prefills* allows constructing multiple *decode-maximal batches* from a single prefill request, maximizing coverage of decodes that can piggyback. Furthermore, the uniform compute design of these batches ameliorates the imbalance between micro-batches, significantly reducing pipeline bubbles.

Our techniques yield significant improvements in inference performance across models and hardware. For the LLaMA-13B model on A6000 GPU, SARATHI improves decode throughput by up to 10×, and accelerates end-to-end throughput by up to 1.33×. For LLaMa-33B on A100 GPU, we achieve 1.25× higher end-to-end-throughput and up to 4.25× higher decode throughput. When used with pipeline parallelism on GPT-3, SARATHI reduces bubbles by 6.29×, resulting in an end-to-end throughput improvement of 1.91×.

1 Introduction

The scaling up of language models [25, 26, 35, 38] has led to an emergence in their abilities [45] in a variety of complex tasks — natural language processing, question answering, code generation, etc. This has led to an explosion in their usage across applications spanning conversational engines [2, 4, 5, 38], search [3, 8, 9, 15, 22], code assistants [1, 7, 16], etc. The significant GPU compute required for inference on these large models, coupled with their widespread usage, has made LLM inference the dominant GPU workload. Optimizing LLM inference has thus become very important and has seen significant interest recently [39, 42, 48].

In this paper, we first analyze a fundamental reason behind the low efficiency of LLM inference. Each LLM inference request goes through two phases – a *prefill* phase corresponding to the processing of the input prompt and a *decode* phase which corresponds to the autoregressive token generation. The prefill phase processes *all* tokens in the input sequence in parallel, leading to high GPU utilization even with a small batch size. For example, on an A6000 GPU, for the LLaMA-13B model, a prefill with a sequence length of 512 tokens

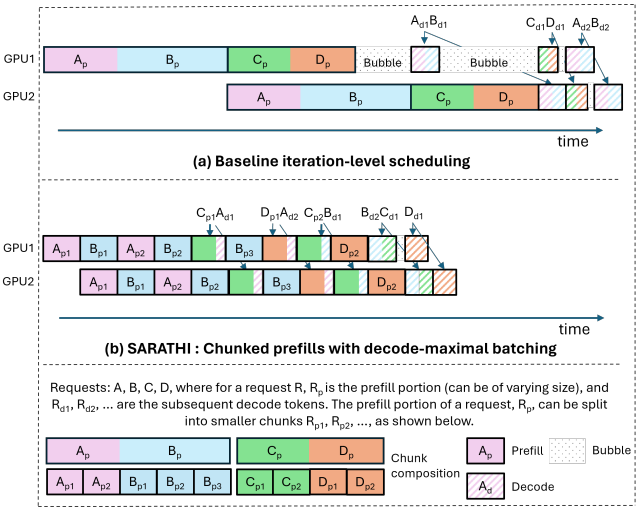


Figure 1: **Example two-stage pipeline parallel schedule.** (a) In prior solutions like Orca [48], pipeline bubbles are common due to varying prompt and decode compute times. Further, decodes are highly inefficient (decode *cost-per-token* is order-of-magnitude higher than Prefill). (b) SARATHI significantly reduces pipeline bubbles and enables more efficient *piggybacked decodes*.

*Work done as intern at Microsoft Research India

saturates GPU compute even at a batch size of just one. The decode phase, on the other hand, processes only a *single* token in each autoregressive pass, resulting in very low GPU utilization at low batch sizes. For example, our experiments reveal that, at small batch sizes, the decode cost per token can be as high as ~ 200 times the prefill cost per token. Moreover, since a request goes through only a single prefill pass, but multiple decode passes (one for each generated token), the overall inference efficiency is significantly impacted.

One strategy to improve LLM decode efficiency is to increase batch size using model parallelism. In servers with high bandwidth connectivity such as NVIDIA DGX A100, tensor-parallelism [43] can enable deployment of an LLM on up to 8 GPUs, thereby supporting large batch sizes and efficient decode. Pope et al. [39] show that tensor parallelism can be scaled up to 256 devices on specialized TPUv4 pods. However, tensor-parallelism at such a large scale can result in poor performance when hyper-clusters are unavailable. In such cases, pipeline parallelism [24, 37] can help increase batch size. Thus, systems like Orca [48] rely on pipeline parallelism to scale LLM inference and adopt the well-known solution of using micro-batches to mitigate pipeline stalls or bubbles [34]. However, as we show in this paper, *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. This creates varying processing times for the different micro-batches, resulting in significant bubbles and wasted GPU-cycles as illustrated in Figure 1(a). Note that the first bubble in the figure is due to varying prompt sizes while the second bubble is due to mismatch between prompt and decode compute times.

In this paper, we present the design and implementation of SARATHI, an efficient LLM inference technique. SARATHI uses *chunked-prefills* and *decode-maximal batching* to address the problems of 1) inefficient decodes and 2) pipeline bubbles. *Chunked-prefills* splits a prefill request into equal compute-sized chunks. Further, SARATHI uses *decode-maximal batching* to construct a batch by using a single prefill chunk and filling the remaining batch with decodes. This hybrid batch provides units of work that are both compute saturating and uniform, thereby addressing the problems of inefficient decodes and pipeline bubbles.

Since *prefill* and *decode* phases have different compute requirements, the key insight of our method is that mixing prefill and decode requests in a single batch can enable uniformly high compute utilization. However, since each request has only a single prefill phase, followed by multiple decode phases (for each generated token), we will not have enough prefill requests to be able to always create a hybrid batch of prefills and decodes. *Chunked-prefills* allows us to construct multiple hybrid batches from a single prefill request, thereby increasing the coverage of decodes that can piggyback with a prefill. In our hybrid batch, the single prefill chunk ensures high GPU utilization, while the decode phase requests 'piggyback' along. Given an average prefill-to-decode token ratio for an LLM application, we select a prefill chunk size that maximizes the overall performance.

The hybrid batches constructed in SARATHI have a *uniform* compute requirement. Thus, when used with pipeline parallelism, SARATHI ensures that the micro-batches are well balanced, which results in a significant reduction in pipeline bubbles as shown in Figure 1(b).

We evaluate SARATHI across different models and hardware — LLaMA-13B on A6000 GPU, LLaMA-33B on A100 GPU, and GPT-3 with 8-way pipeline and 8-way tensor parallelism across a simulated cluster of 64 A100 GPUs. For LLaMA-13B on A6000, SARATHI improves decode throughput by up to $10\times$ and results in up to $1.33\times$ end-to-end throughput improvement. Similarly, for LLaMA-33B on A100, our decode throughput improves by $4.25\times$, and results in a $1.25\times$ end-to-end throughput improvement. When used with pipeline parallelism, SARATHI reduces bubbles by $6.29\times$, resulting in end-to-end speedup of $1.91\times$.

The main contributions of our paper include:

1. *Chunked-prefills* which allows the construction of work units that are compute saturating and uniform.
2. *Decode-maximal batching* which allows inefficient decodes to 'piggyback' with efficient prefills.
3. Application of *chunked-prefills* and *decode-maximal batching* to pipeline parallelism to significantly reduce pipeline bubbles.
4. Extensive evaluation over multiple models, hardware, and parallelism strategies demonstrating up to $1.91\times$ improvement in throughput.

2 Background

We first give an overview of the transformer architecture, followed by a brief discussion of the two phases of LLM inference, and pipeline parallelism.

2.1 The Transformer architecture

Figure 2 shows the architecture of a transformer decoder block. Each decoder block consists of two primary modules: self-attention and feed-forward network (FFN). These modules can be divided into the following six operations: *preproj*, *attn*, *postproj* (within the attention module), and *ffn_ln1*, *ffn_ln2* (within FFN) and *others* (e.g., layer normalization, activation functions, residual connections etc.).

2.2 The prefill and decode phases

Transformer inference begins with the prefill phase that processes all the input tokens of a given batch in parallel. In this phase, the input to a transformer block is a tensor X of shape $[B, L, H]$ where B denotes the batch size, L denotes the

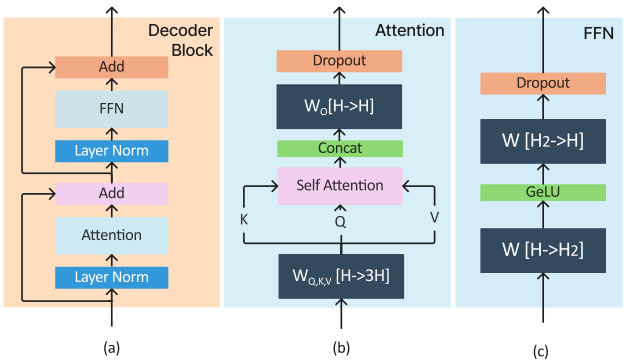


Figure 2: High-level architecture of a decoder block.

Operation	Shapes of tensors		
	Input(s)	Weight(s)	Output(s)
preproj	$[B,L,H]$	$[H,H]$	$[B,L,H]$
attn	$[B,L,H]$	-	$[B,L,H]$
postproj	$[B,L,H]$	$[H,H]$	$[B,L,H]$
ffn_ln1	$[B,L,H]$	$[H,H_2]$	$[B,L,H_2]$
ffn_ln2	$[B,L,H_2]$	$[H_2,H]$	$[B,L,H]$

Table 1: Shapes of the input, weight, and output tensors in a transformer decoder block. B, L and H denote batch size, embedding (aka hidden) size and sequence length (L=1 during decode, except for attention).

sequence length of each request (i.e., the number of input tokens in the given query), and H is the model's embedding size (e.g., 5120 for LLaMA-13B).

Table 1 shows the shapes of input, output, and weight tensors of the various operations. Each transformer block first computes self-attention on a given input X. Typically, multi-head attention is used, but we consider only one head for simplicity of exposition. A linear transformation *preproj* over X (using the weight tensors W^Q, W^K and W^V of shape $[H,H]$) produces the Q, K and V that are commonly known as queries, keys, and values, each of shape $[B,L,H]$. Internally, *preproj* is a single matrix-matrix multiplication of X with a combined weight tensor of shape $[H,3H]$.

Next, the *attn* computation over Q, K and V produces a tensor Y of shape $[B,L,H]$. Finally, *postproj* applies a linear transformation over Y (using weight matrix W_o of shape $[H,H]$), returning a tensor Z of shape $[B,L,H]$.

Next, the FFN module performs two batched matrix-matrix multiplications. In *ffn_ln1*, Z is multiplied with a weight tensor of shape $[H,H_2]$ producing an output tensor of shape $[B,L,H_2]$, which is then multiplied by a weight tensor of shape $[H_2,H]$ in *ffn_ln2* to output a tensor of shape $[B,L,H]$. Here, H_2 refers to the second hidden dimension of the model.

The decode phase performs the same operations as prefill, but only for the *single* token which was generated in the last autoregressive iteration. Thus, the input tensor in decode phase is of shape $[B,1,H]$ (as opposed to $[B,L,H]$ of prefill). Further, the attention computation for each new token depends on the key (K), and value (V) tensors of all prior tokens in the same request. To avoid recomputing K and V of all tokens in every iteration, most implementations cache these values in GPU memory - which is referred to as the KV cache. Note that each token's K and V tensors are of shape $[1,H]$.

2.3 Multi-GPU LLM Inference

As the model sizes of LLMs increase, it becomes necessary to scale them to multi-GPU as well as multi-node deployments [19, 39]. 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 shards the model weights across multiple GPUs freeing up memory to support larger batch sizes. Prior work has employed both tensor-parallelism (TP) [43] (within node) and pipeline-parallelism (PP) [6, 46, 48] (across nodes) for this purpose.

TP shards each layer across the participating GPUs. This splits both the model weights and KV cache equally across GPU workers, leading to linear scaling of per-GPU batch size. However, it comes at a high communication cost due to two all-reduce operations per layer – one in attention computation and the other in FFN [43]. Moreover, since these communication operations are in the critical path, TP is preferred only within a single node connected by high bandwidth interconnects like NVLink. PP is primarily used to facilitate cross-node deployments for very large models, where the model cannot fit within a single node.

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 the advantage of a much better compute-communication ratio compared to TP, as we only need to send activations once for multiple layers of compute. Furthermore, PP requires communication only via point-to-point communication operation, compared to the more expensive allreduces required in TP. Thus, PP is the only viable model-parallelism approach when high-bandwidth connectivity like NVlink is unavailable at cluster-scale. In such settings, the use of PP can help increase the maximum batch size supported in each node by 2-3×, thereby improving LLM inference efficiency.

3 Motivation

In this section, we show that LLM inference is inefficient for two main reasons: (1) the decoding phase is memory-bound, and (2) the use of pipeline parallelism leads to significant pipeline bubbles for LLMs. Together, these factors lead to poor GPU utilization for LLM inference.

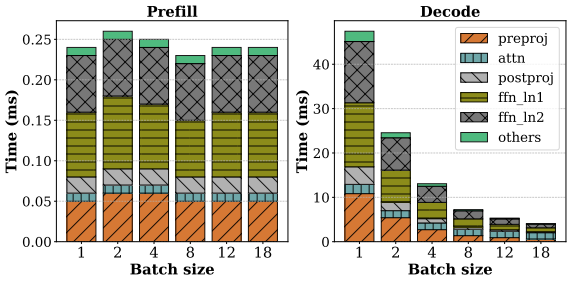
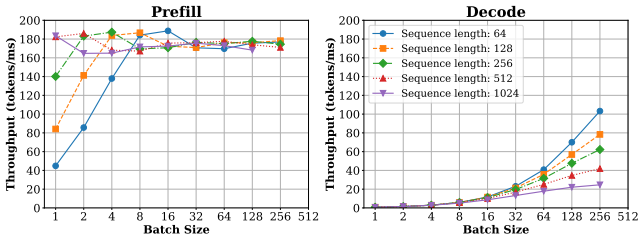
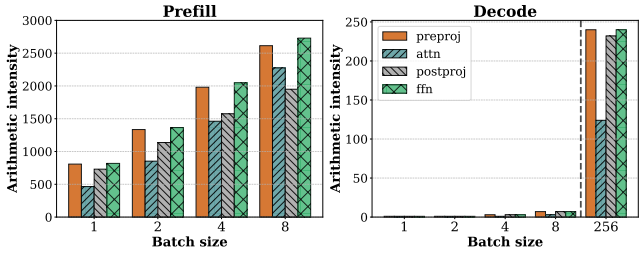


Figure 3: Per-token prefill and decode time with different batch sizes (sequence length = 1024) for LLaMa-13B on A6000 GPU. Prefill saturates GPU compute even at batch size of 1 and results in almost constant per-token time across batch sizes. Decode under-utilizes GPU compute and costs as much as 200× prefill for batch size 1. The incremental cost of linear operators for decode is almost zero as batch size increases. The attention cost does not benefit from batch size as it is memory-bound.



(a) Throughput of a single layer of LLaMA-13B on A6000 GPU.



(b) Arithmetic intensity with 1K sequence length (per-request).

Figure 4: Impact of the arithmetic intensity (bottom) on the throughput (top) of prefills and decodes for LLaMA-13B on A6000 GPU.

3.1 Analyzing Prefill and Decode Throughput

Figure 3 shows the per-token cost of each of the six transformer operations (§2.1) for prefill and decode at various batch sizes for a fixed sequence length (prefill+decode) of 1024. First, we observe that prefill has almost constant per-token cost across various batch sizes, indicating that prefill saturates the GPU even at batch size of 1. Second, we see that decode behaves very differently from prefill as the per-token cost reduces significantly when the batch size increases. Third, we see that the *decode cost per-token is 200×, 100×, and 16.7× that of prefill at batch size of 1, 2 and 18, respectively*. Thus, it is clear that optimizing decodes is critical for efficient LLM inference. Finally, we see that the operations under *others* contribute less than 5% of the overall runtime of the transformer block. Hence, we focus on only optimizing the five major operations and ignore others.

Figure 4a shows the throughput of the prefill and decode stages for different batch sizes (B) and sequence lengths (L). We observe that the throughput of the prefill phase saturates at about 180 tokens/millisecond when $B \times L \geq 512$: e.g., a single prefill request can achieve peak throughput at $L \geq 512$. In contrast, the decode throughput increases linearly with small batch sizes. To further understand the saturation point of decode phase, we profile a single layer as opposed to the 40 layers of the full model. This enables us to fit 40× larger batches on the GPU due to the reduced memory footprint of model weights and KV caches. We find that decode saturates at a much larger batch (e.g., 256 with 1024 sequence length). Such large batches are infeasible to run with the full model.

To explain this behavior, we profile the arithmetic intensity of individual operations: arithmetic intensity captures the amount of compute per memory read/write that can be used to distinguish between compute-bound and memory-bound operations. Figure 4b shows the arithmetic intensity of each operation separately for prefill (left) and decode phases (right). As shown, in prefill phase, all operations have high arithmetic intensity, even at a batch size of *one*. On the other hand, the arithmetic intensity of these operations drop by more than two orders of magnitude in the decode phase. Only at a very large batch size of 256, the decode phase starts becoming compute-intensive. However, scaling up the batch size to such high values is infeasible due to the KV-cache footprint of each request. For instance, we can fit a maximum batch size of 18 requests at a sequence length of 1K for the LLaMA-13B model on an A6000 GPU. Therefore, in the range of batch sizes that are practical today, the decode phase remains memory-bound.

The difference between the throughput scaling of these two phases stems from the fact that the prefill phase computes (batched) *matrix-matrix multiplications* as opposed to the *vector-matrix multiplications* of the decode phase. It is well-known that kernels with arithmetic intensity above a GPU's FLOPS:MemBandwidth ratio are compute-bound and can be executed efficiently [11]. In contrast, kernels with a lower arithmetic intensity fail to utilize GPUs well due to being memory-bound.

3.2 Pipeline Bubbles in LLM Inference

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). PP splits a model layer-wise, where each GPU is responsible for a sub-

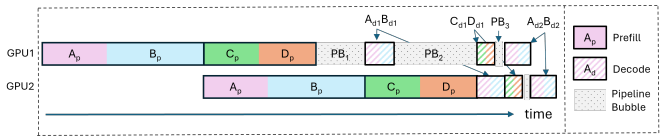


Figure 5: **Pipeline bubbles in LLM inference** A 2-way PP iteration-level schedule [48] across 4 requests (A,B,C,D) shows the existence of pipeline bubbles due to non-uniform batch execution times.

set of layers; compared to TP which shards each layer across the participating GPUs. As discussed in §2.3, compared to TP, PP has a much better compute-communication ratio and does not require expensive interconnects.

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 thus commonly employed in PP training jobs to amortize the bubbles across the multiple micro-batches forming a batch [24, 34, 37].

Unlike training, since inference jobs only do forward passes and do not have backward passes, one might expect that the use of micro-batches will fully avoid pipeline bubbles during inference. In fact, prior work on transformer inference, such as, FasterTransformer [6] and FastServe [46] use micro-batches and do not consider the problem of bubbles with PP.

Orca [48] suggests that the use of iteration-level scheduling eliminates bubbles in pipeline scheduling (see Figure 8 in [48]). However, as we show in this paper, even with iteration-level scheduling of requests, 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 5). 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 accumulated context length (KV cache length) varies across requests. These pipeline bubbles are wasted GPU cycles and directly correspond to a loss in serving throughput with pipeline parallelism. If we can ensure that each micro-batch performs uniform computation, we can mitigate these pipeline bubbles.

3.3 Insights

Our experiments show that the prefill and decode stages have very different compute utilization patterns – prefill can saturate GPU compute even with a single request, while decodes require a large batch size to be compute-efficient. However large batches are impractical due to their high KV cache footprint. Such disproportionate resource utilization implies that for every request, there are phases of high compute utilization due to efficient prefills, followed by a potentially long tail of inefficient decodes which results in poor overall GPU utilization. Furthermore, the non-uniformity in compute times across micro-batches leads to pipeline bubbles, resulting in inefficient pipeline parallel multi-GPU deployments.

This observation leads us to our key insight that it is possible to construct uniformly compute-intensive batches by (1) slicing a large prefill request into smaller compute-efficient and uniform chunks using *chunked-prefills* and (2) creating a hybrid batch of a prefill chunk and piggybacking decodes alongside this chunk. Consequently, creating such uniform and compute-intensive batches ensures high GPU utilization throughout, as well as, minimizes pipeline bubbles in multi-GPU deployments by eliminating the runtime variance across micro-batches in different stages of the pipeline.

4 SARATHI: Design and Implementation

In this section, we describe the design and implementation of SARATHI, which employs two techniques - *chunked-prefills* and *decode-maximal batching* to improve the performance of LLM inference.

4.1 Overview

Conventional inference engines like FasterTransformer [6] perform request-level inference scheduling. They process batches at request granularity; i.e., the they pick the next batch of requests to execute on the model replica only when all the requests in the current batch complete. While this reduces the operational complexity of the scheduling framework, it is inefficient in its use of resources. Shorter requests in a batch have to be padded to match the length of the longest request, and thus does wasteful work instead of exiting early. Alternatively, iteration-level scheduling has been proposed in more recent systems like Orca [48], vLLM [20], and HuggingFace TGI [17], where depending on the predetermined batch size b, requests can dynamically enter and exit a batch.

However, today's iteration-level scheduling systems do not pay attention to the requests that comprise the batch, and the varying execution time between batches. Specifically, a batch could comprise of requests only in the prefill phase, requests only in the decode phase, or mixed requests consisting of a few prefills and decodes, with the only constraint that the batch size is b at all times. As discussed in §3.3, such batch formation results in non-uniform units of compute, resulting in periods of bursty resource utilization, and pipeline bubbles. SARATHI tackles this challenge by introducing two key techniques: *chunked-prefills* and *decode-maximal batching*.

Prefill 过程中，每一个 token 的计算都是 “独立” 和 “可并行” 的，因为可以通过 attention mask 来看到所有前面的 tokens 和 mask 掉后面的 tokens，所以可以 chunked

	k0	k1	k2	k3
q0	1	-	-	-
q1	1	1	-	-
q2	1	1	1	-
q3	1	1	1	1

attention mask during first chunk prefill

	k0	k1	k2	k3	k4	k5	k6	k7
q4	1	1	1	1	1	-	-	-
q5	1	1	1	1	1	1	-	-
q6	1	1	1	1	1	1	1	-
q7	1	1	1	1	1	1	1	1

attention mask during second chunk prefill

	k0	k1	k2	k3	k4	k5	k6	k7	k8	k9	k10	k11
q8	1	1	1	1	1	1	1	1	1	-	-	-
q9	1	1	1	1	1	1	1	1	1	1	-	-
q10	1	1	1	1	1	1	1	1	1	1	1	-
q11	1	1	1	1	1	1	1	1	1	1	1	1

attention mask during third chunk prefill

Figure 6: Example of how attention mask is set across different chunk prefill iterations in SARATHI (q and k represent “query" and “key" tokens, respectively). The attention mask for v (“values") is set similarly.

4.2 *Chunked-prefills*

Chunked-prefills is a prefill splitting mechanism hinged on two key insights. First, for a given model and GPU, increasing the number of prefill tokens shows diminishing returns in throughput beyond a certain point as shown in Figure 4a. For instance, the Llama-13B model achieves peak prefill throughput on an A6000 GPU when the number of prefill tokens is 512 or higher. At a chunk size of 256, we see a marginal reduction of 12.5% in the peak throughput. Further, as the size of the hidden dimension in the model increases, the chunk size needed to saturate the GPU compute drops; for example, the throughput of a single layer of GPT-3 (hidden size = 12288) peaks at a chunk size of 256 on an A100 GPU. This implies that a compute-saturating batch can be formed with a carefully sliced prefill chunk. Second, in many practical scenarios, the size of prefill is reasonably large, ranging from 1K – 4K in production workloads, thereby opening the doors for chunking a prefill request into smaller units of compute.

Implementing *chunked-prefills* requires carefully setting the attention mask. If a request’s input prompt of say size 1K is split into four chunks of size 256 tokens each, we need to ensure that the attention masks are appropriately set for every subsequent prefill chunk until the end of the prompt. For ease of exposition, using an example of chunk size of four, Figure 6 shows how SARATHI progressively sets the attention mask for every successive chunk of a prefill prompt in three consecutive iterations: each query token q_i can peek into the keys (and values) of all the tokens preceding it, but not the ones that follow. Setting the attention mask this way ensures that *chunked-prefills* computation is mathematically equivalent to the full prefill.

Overhead of *chunked-prefills*. Splicing the input of a prefill sequence into multiple smaller chunks has two potential sources of overhead. First, the arithmetic intensity of *chunked-prefills* computation decreases as the chunk size becomes smaller. Therefore, smaller chunks can affect prefill efficiency due to low GPU utilization. However, this can be addressed easily with a one-time profiling of the prefill throughput for various chunk sizes on a given model-hardware combination and expected workloads and a chunk size can be chosen such that the end-to-end throughput of the model is maximized.

Second, *chunked-prefills* pose a slight overhead in attention computation due to repeated memory accesses of the KV cache of a request’s tokens from prior chunks. While every *chunked-prefills* operation until the end of the prompt will perform the same number of computations for FFNs, the attention kernel in every subsequent chunk after the first will have to reread all the KV pairs of the prior tokens from the GPU memory, as shown in Figure 6. For example, if a prefill sequence is split into N chunks, then the first chunk’s KV cache is loaded N times, the second chunk’s KV cache is loaded $N-1$ times, and so on. However, the overhead due to increased attention time does not significantly affect the end-to-end prefill efficiency because attention computation is a small fraction of the overall forward pass time as seen in Table 2. We present a detailed analysis of the overheads of *chunked-prefills* in §5.4.

4.3 Decode-Maximal Batching

Harnessing the benefits of *chunked-prefills* requires us to carefully construct a hybrid batch consisting of a mix of prefill and decode tokens, so as to maximize compute utilization and ensure uniform compute time across all batches. We propose *decode-maximal batching* to alleviate the imbalance in compute and memory utilization in iterative scheduling by exploiting the idea of *chunked-prefills*.

In *decode-maximal batching*, we construct a batch by using a single prefill chunk and piggybacking the remaining slots with decode tokens. This hybrid batch provides us with units of work that are both compute saturating and uniform. We now discuss how we construct a hybrid batch to achieve maximum efficiency.

4.3.1 Piggybacking decodes with prefills

To piggyback decodes with a prefill, we need to take care of two things. First, we need to identify the maximum possible batch size of decodes that can be piggybacked and also identify the number of prefill tokens that comprise the prefill chunk. Second, in order to actually utilize the GPU-saturating prefill computation of the hybrid batch to make the decodes efficient, we need to *fuse* the linear operation computations for the prefill chunk and decodes of the batch into a single operation.

Decode batch. The maximum decode batch size to be piggybacked with a prefill chunk is determined based on the

Prefill computation 没咋变化甚至还更快的原因是 chunked prefill 让 pipeline bubbles 变得更少

Decode per-token cost 下降是因为多个 requests 的 decode batch 到一块儿减少了 weight read，和 chunked prefill 并不强相关

Batching Scheme	Operation(s)		Total Time	Per-token Time	
	Linear	Attn		Prefill	Decode
Prefill-only	224.8	10	234.8	0.229	-
Decode-only	44.28	5.68	49.96	-	**12.49**
Decode-maximal	223.2	15.2	238.4	0.229	**1.2**

Table 2: **Per-token prefill and decode time (in ms)** For LLaMA-13B on A6000 GPU, the rows show operation times for 1) prefill-only requests of prompt size 1024 of batch size 4, 2) decode-only batch size of 4 with sequence length 1024, and c) a mixed batch of a single 1021 prefills and 3 decodes. *Decode-maximal batching* reduces the decode time per token by an order of magnitude.

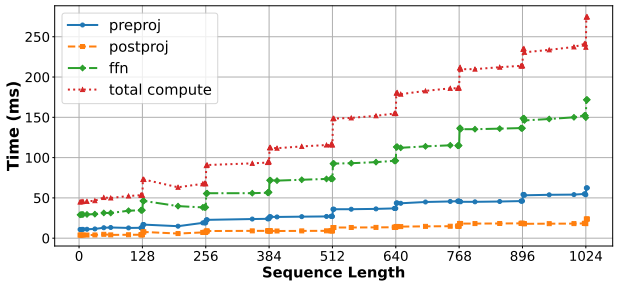


Figure 7: The effect of tile quantization on the runtime of one iteration of LLaMA-13B on A6000 GPU.

available GPU memory (M_G), the model's parameter memory requirement per GPU (M_S), and the maximum sequence length L that the model supports. The total of prefill (P) and decode (D) tokens per request cannot exceed this maximum sequence length. Assuming the memory required per pair of K and V for a token is m_{kv}, the maximum permissible batch size B is determined as follows

$$B = \lfloor \left(\frac{M_G - M_S}{L * m_{kv}} \right) \rfloor$$

In the baseline scheme, decode-only batches can be of size at most B. In SARATHI, the number of decodes can be at most $B-1$ as they piggyback along with one prefill chunk (the prefill's KV cache also needs to be in GPU memory until its corresponding decode iterations begin).

In *decode-maximal batching*, we fuse all the linear operations, while letting the attention computations for the prefill and decodes happen separately. The attention operation for decode requests is batched together, while the attention in prefill chunk is processed separately.

Decode efficiency. Recall that the prefill and decode phases follow the same computation path, i.e., the linear operations use the same weight tensors in both the prefill and decode phases. However, compared to prefill, a decode iteration consists of only a few input tokens (equal to the batch size). Therefore, most of the computation time in baseline decoding is spent fetching model weights from GPU's global memory.

In contrast, *decode-maximal batching* computes over the decode tokens using matrix matrix multiplications, by combining decode tokens with the prefill tokens in a single matrix multiplication operation. This, effectively eliminates the need to load the model weights separately for decoding — i.e., once the model weights are fetched for prefills, they are also reused for decoding. As a result, *decode-maximal batching* converts decoding from being in a memory-bound phase to being in a compute-bound phase. This way, decodes, when piggybacked with prefills come at a marginal cost in SARATHI (note that the attention cost remains unchanged).

To illustrate the various costs involved through an example, Table 2 compares the runtime of one iteration of *decode-maximal batching* with that of the baseline scheme that computes prefill and decode iterations separately. With baseline batching, a decode-only iteration spends 12.49 milliseconds per token. In contrast, per-token decode time is only 1.2 milliseconds with *decode-maximal batching*. This shows that piggybacking decodes with prefills can improve decode throughput by up to an order to magnitude.

4.4 Identifying the ideal chunk size

An important design consideration in SARATHI is *how to pick the most suitable chunk size*. A straightforward choice is to pick the smallest chunk size that saturates a model's prefill throughput. However, we find that this strategy is not the most efficient in many cases.

To demonstrate the importance of chunk size, we introduce a simple notation "P:D ratio" that is computed as the ratio of the number of prefill tokens to the number of decode tokens in a given batch. For example, a P:D ratio of 10 implies that the number of prefill tokens is 10 times that of decode. For a fixed P+D, a lower value of P:D ratio means that there are more decode tokens in a batch compared to one with a higher value of P:D ratio.

The size of prefill chunks in SARATHI impacts the number of decodes that can be piggybacked using *decode-maximal batching*. For example, consider a batch size of four requests (where one request is in the prefill phase and three are in the decode phase) and a chunk size of 128. A prefill of size P will then yield P/128 prefill-chunks, allowing P/128 $\times 3 \approx P/42$ decodes to piggyback. Thus, in this case, when the P:D ratio is greater than 42, it allows us to overlap all decodes with prefills. Similarly, if the chunk size is 256, then all decodes can be piggybacked when the P:D ratio is greater than 84. Therefore, a lower chunk size can help piggyback more decode tokens for a given prefill sequence.

Note that decoding time increases as the the P:D ratio goes down. Therefore, beyond a certain point, optimizing decodes becomes more important than executing prefills at peak efficiency. For example, if the prefill and decode phases consume 10% and 90% of the total time, respectively, then

even a 5× overhead in prefills is acceptable if the decodes can be optimized by 2× or more.

To sum it up, identifying a suitable chunk size involves a trade-off: smaller chunks piggyback more decodes but at the expense of lower prefill efficiency whereas larger chunks are prefill efficient but piggyback fewer decodes. Therefore, the ideal chunk size depends on the expected P:D ratio and the split between prefill and decode times for a given application.

The tile quantization effect. Additionally, we observe an intricate detail related to the chunk size. 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 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 [11].

Notice that the time to compute a prefill sequence suddenly increases when the sequence length is just higher than a multiple of 128 (tile size in our experiments). For example, as shown in Figure 7, doubling the sequence length from 128 to 256 tokens increases iteration time by 27% — from 55ms to 69.8ms. However, adding only a single token further increases the iteration time to 92.33ms — a dramatic 32% increase due a only one additional token. This shows that the GPU is most efficient at matmuls when the sequence length is a multiple of the tile size.

Therefore, selecting the ideal chunk size is a two-fold decision. First, pick a chunk size based on the desired prefill efficiency for the given workload. Next, ensure that the sum of chunk size and the number of piggybacked decode tokens is a multiple of the tile size. This ensures that the relevant matrix dimension of the fused operations stays a multiple of the tile size. For example, if the chosen chunk size is 256, the tile size is 128, and the maximum permissible batch size is B, then, the prefill chunk size should be $256-(B-1)$.

4.5 Implementation

We implement SARATHI on the nanoGPT codebase [12] with support for both *chunked-prefills* and *decode-maximal batching*. To compare against Orcas's iteration-level scheduling, we use our mixed batching mechanism, with no constraint on the number of prefills allowed per batch. This ensures that there is no discrepancy in results between the baselines and SARATHI due to differences in implementation. To compute the attention operation, we use xformers implementation [21] as in our setup, it outperformed PyTorch 2.0's in-built attention implementations: i.e., flash attention, memory-efficient attention, and math attention kernels. To avoid allocating memory for KV caches in each decode iteration, we pre-allocate the KV cache as per the maximum sequence length for each experiment and update respective KV pairs in place when required.

Model	GPU	Num GPUs	Per-GPU Mem(GB)	Mode
LLaMA-13B	A6000	1	48	Deployment
LLaMA-33B	A100	1	80	Deployment
GPT-3	A100	64	80	Simulation

Table 3: Models, GPUs, and mode of evaluation.

We support different model configurations in our codebase to evaluate SARATHI over different model and hardware combinations. For example, to evaluate LLaMA-13B, we set the number of layers and attention heads to 40, and hidden size to 5120. For LLaMA-33B, we use 60 layers, 52 attention heads, and hidden size of 6656. For GPT-3, we use 96 layers, 96 attention heads, and hidden size of 12288. The configurations are as per the publicly available architectural parameters of these models [10, 14].

5 Evaluation

We evaluate SARATHI on a variety of models and GPUs using physical deployments for single GPU experiments and profile-driven simulations for large-scale experiments as shown in Table 3. Our evaluation seeks to answer the following questions:

1. What is the impact of SARATHI on the throughput of decodes as well as the end-to-end throughput of LLMs? In addition, what is the impact of varying sequence lengths, batch sizes, and P:D ratios (§5.1)?
2. How does SARATHI compare to existing iteration-level scheduling mechanisms like Orca (§5.2)?
3. What is the impact of our techniques on GPU bubbles and the throughput of pipeline-parallel models (§5.3)?
4. What are the overheads of *chunked-prefills* (§5.4)?

5.1 Evaluation on a Single GPU

In this section, we measure the decode speedup and the end-to-end throughput of SARATHI, on a single GPU, against that of the baseline which executes the prefill and decode stages separately via prefill-only and decode-only batches. Further, we examine the effects of varying $P:D$ ratio (ratio of prefill to decode tokens), sequence lengths (total tokens per request — $P+D$), and batch sizes on the overall throughput.

5.1.1 Decode speedup

We first show the impact of our techniques on decode phase throughput that we calculate based on the average time spent on decoding one token. For the baseline system, we compute the average decode time per token by dividing the time to process one decode iteration by the batch size. In SARATHI,

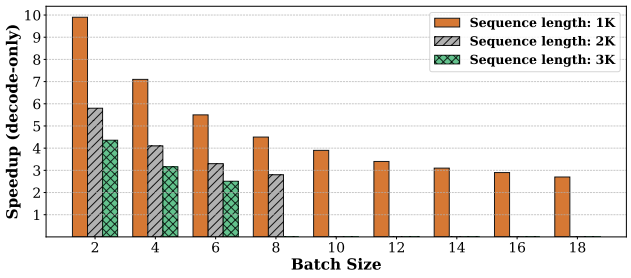


Figure 8: Decode-only speedup with SARATHI on an A6000 GPU with LLaMA-13B (chunk size = 256).

Model (GPU)	Sequence Length	Batch Size	P:D Ratio	Decode Speedup	Throughput Gain
LLaMA-13B (A6000)	1K	6	50:1	5.45×	1.33×
	2K	6	50:1	3.26×	1.26×
	3K	6	50:1	2.51×	1.22×
LLaMA-33B (A100)	1K	10	28:1	3.83×	1.25×
	2K	5	63:1	4.25×	1.22×
	3K	3	127:1	3.51×	1.14×

Table 4: Peak throughput gains with SARATHI for different sequence lengths with two different model-GPU combinations (chunk size = 256).

where decodes are piggybacked, for a batch with $p+d$ tokens, where p denotes the prefill chunk size and d denotes the decode batch size, we find the difference in runtime between the *decode-maximal* batch and a prefill-only batch of prefill size p, and attribute the difference in time as the marginal decode time for a batch of d requests. This marginal decode time is then used to compute the decode time per token.

Figure 8 plots the results for a chunk size of 256 for LlaMa-13B on A6000 GPU, as we vary the batch size, up to the respective maximum value that fits, for three different prefill sequence lengths. We observe that *chunked-prefills* improves decode efficiency by up to an order of magnitude over baseline. Decode throughput of SARATHI is higher due to *decode-maximal batching* that computes decode tokens with matrix-multiplications, allowing reuse of the model weights — for both prefills and decodes — once they are fetched from the GPU's global memory.

We observe that our decode speedup reduces as we increase the batch size or sequence length. This behavior is expected for the following reasons: (1) decodes in the baseline system become more efficient as the batch size increases, and (2) the cost of attention increases quadratically with the sequence length: since all our improvements come from optimizing the linear operations, a higher attention cost reduces our scope for improvement. However, our decode throughput improvement is still significant in all cases ($2.8\times - 10\times$).

5.1.2 Peak throughput gains with SARATHI

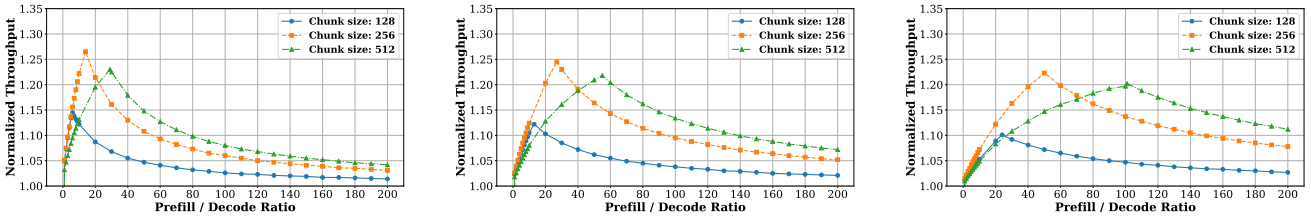
Table 4 shows the peak throughput gain that SARATHI achieves over the baseline. To demonstrate the generality of our techniques, we evaluate SARATHI on two model-GPU combinations: (1) LLaMA-13B on an A6000 GPU and (2) LLaMA-33B on an A100 GPU. Further, we investigate the peak throughput gain with varying sequences of length 1K, 2K and 3K. Table 4 shows the batch sizes and P:D ratios where we achieve the maximum speedup.

In the best case, our techniques improve the end-to-end throughput by as much as $1.33\times$ for LLaMA-13B and up to $1.25\times$ for LLaMA-33B. We observe that the speed up is relatively higher on the A6000 GPU as compared to the A100 GPU. This is due to the higher FLOPs/MemBandwidth of the A100 GPU compared to the A6000 GPU (≈ 156 vs. ≈ 53, ignoring GPU caches). Therefore, we require a higher chunk size on the A100 GPU (or a model with a higher embedding size) to avoid losing the prefill efficiency. However, SARATHI still consistently outperforms the baseline by $1.14\times$-$1.25\times$ on the A100 GPU. These results show that piggybacking decode tokens with prefill chunks is useful across a wide range of models and hardware. We note that although we improve decode efficiency by up to an order of magnitude, the end-to-end speedups and in turn monetary savings in inference cost are in the order of 25%. This is because our technique only improves decodes and not prefills.

5.1.3 Effect of varying $P:D$ ratio

In this subsection, using various sequence lengths and chunk sizes, we investigate the effect of varying $P:D$ ratios on the end-to-end inference throughput to cover a wide range of application scenarios. $P:D$ ratio is an important parameter for these experiments: a lower $P:D$ ratio indicates that a request constitutes more decode tokens compared to other requests with a higher $P:D$ ratio. Although a lower $P:D$ ratio implies that decodes will constitute a larger fraction of the inference cost and thus SARATHI will have more surface area of attack, however, it also means there will be fewer prefill chunks for piggybacking decodes. This trade-off results in a behavior where the improvement from SARATHI peaks at a particular $P:D$ ratio and then tapers off on either side. We discuss this in more detail below.

Figure 9 plots the results of our experiments. We find that the peak efficiency of our techniques occurs at different $P:D$ ratios for different prefill chunk size and batch size scenarios. If C is the chunk size and B is the batch size, then we can show that this peak will occur when the decodes perfectly piggyback with the prefill chunks. This occurs when the number of prefill chunks ($=P/C$) is the same as the required number of decode iterations ($=D/(B-1)$), i.e., when $P:D = C/(B-1)$. For example, using a chunk size of 256 at batch size of 18, SARATHI achieves the peak throughput improvement of $1.27\times$ at $P:D = 14$ ($\approx C/(B-1) = 256/17$)



(a) Sequence length = 1K, batch size = 18. (b) Sequence length = 2K, batch size = 10. (c) Sequence length = 3K, batch size = 6.

Figure 9: Normalized throughput (tokens/ms) for LLaMa 13B on A6000 GPU with different sequence lengths, P:D ratios, and chunk sizes.

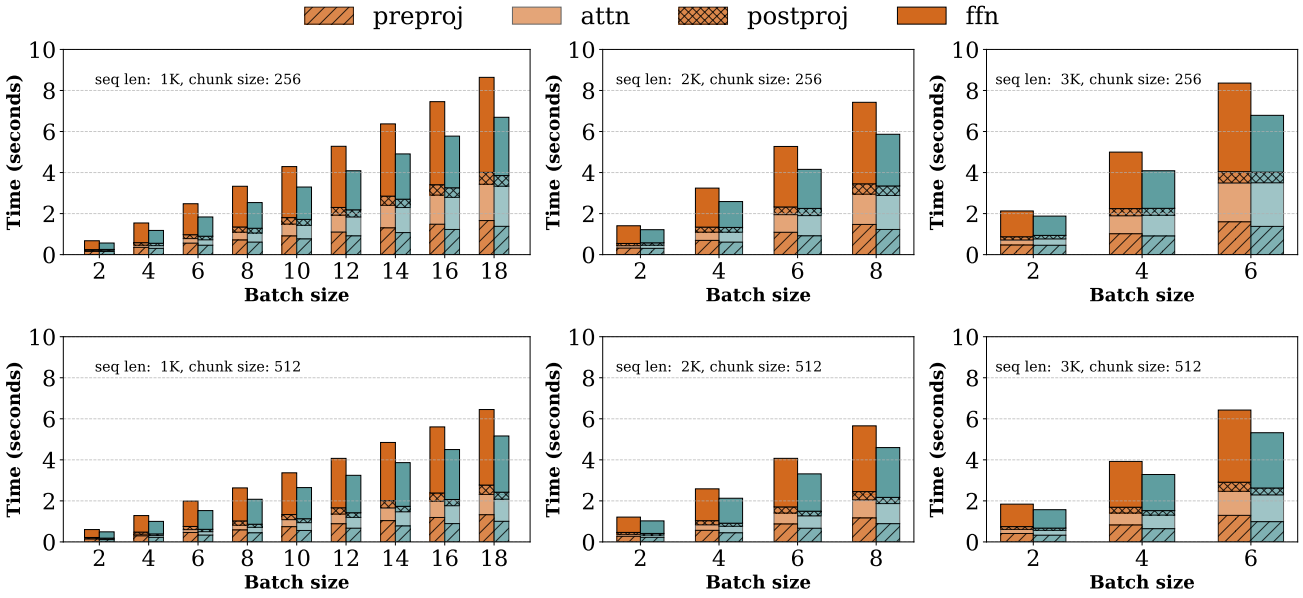


Figure 10: Breakdown of total time spent on different operations for LLaMa 13B on A6000 GPU with varying sequence lengths and batch sizes, using prefill chunk sizes of 256 (top half) and 512 (bottom half). Orange and blue bars represent baseline and SARATHI, respectively.

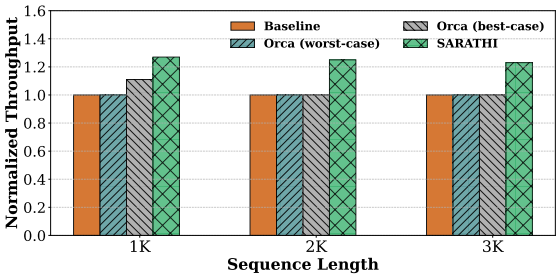
for sequence length of 1K as shown in Figure 9a. Using the chunk size of 512 for sequence length=1K at batch size of 18 also provides significant gains of up to 1.23× at $P:D = 28$ ($\approx C/(B-1) = 512/17$) whereas the gains are much lower with a chunk size of 128. While smaller chunks provide more opportunity to overlap decodes, splitting prefills into very small chunks leads to lower arithmetic intensity i.e. less efficient matmuls and higher overheads (due to multiple reads of KV cache), resulting in reduced end-to-end performance. Thus we obtain a much higher throughput with chunk size of 256/512 compared to the smaller chunk size of 128. Note that the peak gains occur at a higher value of $P:D$ ratio when using a larger chunk size.

We achieve peak performance when inference is not entirely dominated by either prefills or decodes (in other words, when the $P:D$ ratio is balanced). Such a state allows us to overlap prefills and decodes efficiently for longer. Otherwise, SARATHI either runs out of prefill tokens (if $P:D$ is low) or decode tokens (if $P:D$ is high). In these cases, SARATHI can switch to a different chunk size, or operate similar to the standard baseline processing prefill-only or decode-only batches. However, note that despite this variation, our improvements are still around 10% over a large range of $P:D$ ratios.

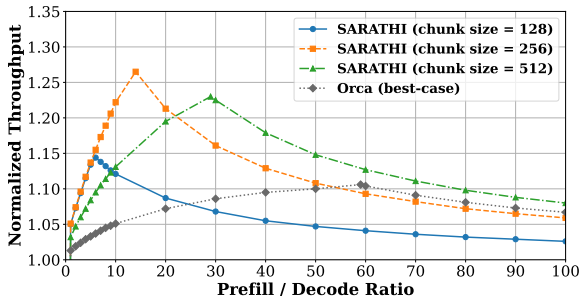
5.1.4 Effect of varying the batch and chunk sizes

In this section, we dive deeper to investigate the performance of SARATHI by varying the batch sizes and chunk sizes for each sequence length. In all these experiments, we focus on execution scenarios where the $P:D$ ratio is balanced i.e., when $P:D = C/(B-1)$ and all decode tokens are perfectly piggybacked with prefills. This allows us to measure the peak performance of our system.

Figure 10 shows the results for these experiments. For each



(a) Varying sequence lengths (chunk size=256 for SARATHI). We choose the maximum batch size which fits for the sequence length (18, 10 and 6 for 1K, 2K and 3K sequence lengths, respectively)



(b) Varying P:D ratio (sequence length=1K, batch size=18).

Figure 11: Comparison with iteration-level scheduler Orca for LLaMa 13B on A6000 GPU.

configuration of sequence length and chunk size, we show the effect of varying batch sizes. Further, for each run, we also show the runtime across different operations i.e., *preproj*, *attention*, *postproj*, and *ffn*.

Note that *decode-maximal batching* batches the prefill and decode tokens in linear operations to improve compute utilization. Therefore, the linear operations see a significant runtime reduction of up to 1.6× (see *ffn* runtime in the first row) compared to the baseline. However, note that the magnitude of improvement also depends on the $P:D$ ratio (in other words, it depends on what fraction of time is spent in decodes). For example, using a chunk size of 256 doubles the number of decodes that can be piggybacked compared to using 512 as the chunk size. Therefore, in the optimal configurations ($P:D=C/(B-1)$), for chunk size of 256, decodes constitute a higher fraction of total runtime, compared to the optimal configuration when chunk size is 512. Therefore, our throughput gains are higher when using chunk size of 256.

We also observe that different linear operations see different speedups using our technique. Linear computation in the *ffn* module sees the highest runtime reduction of 1.3×-1.6×. In contrast, the runtime reduction for *preproj* and *postproj* is 1.05×-1.38×. For small batch sizes, we find that most of the throughput improvement is due to the higher efficiency of *ffn* computation in *decode-maximal batching*.

5.2 Comparison to Iteration-level Scheduling

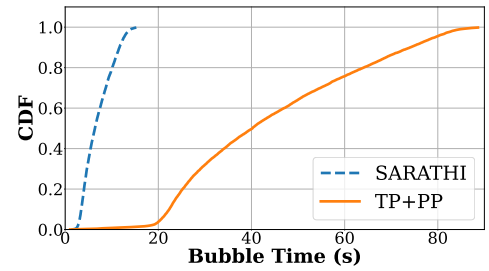
In our evaluation thus far, we have considered a baseline system that processes prefill-only or decode-only batches at a time. This is how popular frameworks like FasterTransformer deploy transformer models. In contrast, Orca's iteration-level scheduling [48] can add (or remove) a request to (or from) a running batch at the granularity of individual iterations.

Iteration-level scheduling affects GPU utilization as well: when requests arrive or depart at different times, some prefills (of newly arriving requests) automatically overlap with the decodes (of already running requests). Therefore, we expect that iteration-level scheduling would do better than the baseline — at least in some cases. However, we emphasize that the overlap between prefills and decodes is more of a side-effect in iteration-level scheduling and its behavior can vary significantly depending on the size and arrival or departure time of requests. Even more importantly, current approaches to iteration-level scheduling submit the entire input sequence of a request in a single prefill phase. This significantly limits the opportunity of piggybacking decode tokens with prefills.

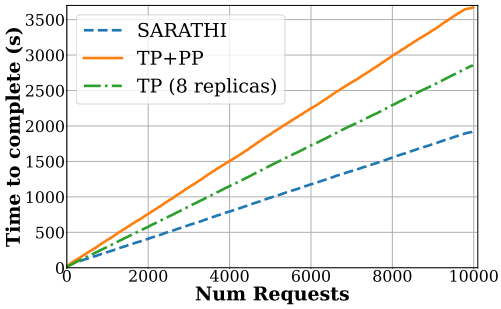
To understand the effect on overall throughput, we evaluate the state-of-the-art iteration-level scheduler, Orca [48], in two scenarios: its best-case and worst-case. In the best case, Orca scheduling overlaps the *full* prefill of *one* new request with the ongoing decodes. In the worst-case, all the requests begin and end at the same time. In the latter case, Orca scheduling behaves similar to our earlier baseline where there is no overlap between the computation of prefill and decode tokens. Note that in the average case of Orca, there could be more than one *full* prefill (corresponding to multiple requests) overlapping with some decodes – this would further limit Orca's ability to piggyback decodes tokens with prefills.

Figure 11 shows our results for these experiments. First (Figure 11a), we show results for the optimal choice of $P:D=C/(B-1)$, where $C=256$ and B is the maximum batch size that fits for the sequence length. As expected, worst-case Orca scheduling performs similar to the baseline. We find that, for a small sequence length of 1K, the best-case Orca scheduling achieves 1.11× higher throughput. This is due to the incidental overlapping of the prefill and decode requests in the best-case schedule. However, as sequence length increases, the performance of best-case Orca scheduling drops close to the baseline. This is an artifact of our choice of $P:D=C/(B-1)$. As we increase sequence length, the batch size B reduces, resulting in a higher optimal $P:D$. Since Orca submits the entire input sequence as a single prefill request, a higher $P:D$ means that it soon runs out of the prefill tokens, at which point it processes the remaining decode tokens similar to the baseline, making even the best-case version inefficient. SARATHI consistently outperforms with overall throughput gains of 1.27×, 1.25× and 1.23× for the three sequence lengths.

Another aspect to consider in iteration-level scheduling is



(a) Comparison of bubble time



(b) End-to-end request completion time

Figure 12: Impact of SARATHI on pipeline bubbles (top) and request completion times (bottom) for GPT-3 deployed on DGX A100(s) in simulation.

the effect of variable sequence lengths on request latencies. Since the prefill time increases with the length of the input sequence, adding a longer prefill sequence in a running batch can delay the ongoing decodes, which in turn increases the latency of these ongoing requests in Orca scheduling. SARATHI avoids this due to the use of smaller chunk prefills.

Next, we evaluate the throughput gains at different $P:D$ ratios for different chunk sizes in Figure 11b. We consider only sequence length of 1K for this experiment as the best-case Orca baseline achieves maximum performance in this regime. Note that best-case Orca scheduling can be considered a special case of SARATHI, where the chunk size, C, is set to the maximum sequence length. As can be seen, the optimal $P:D$ shifts to the right as chunk-size increases. SARATHI with chunk size of 256 performs the best in lower $P:D$ regimes, reaching a peak throughput gain of 1.27× compared to baseline. SARATHI with chunk size of 512 consistently outperforms Orca best-case and performs overall best in the higher $P:D$ regime, reaching a peak throughput gain of 1.23×. In comparison, Orca best-case has much flatter gains and reaches a peak throughput gain of 1.11× at a much higher $P:D$.

5.3 Pipeline Parallelism with SARATHI

Next, we evaluate how SARATHI reduces pipeline bubbles in a multi-GPU pipeline-parallel setup and subsequently impacts the overall runtime of inference jobs. For this experiment, we report evaluations in a carefully simulated environment.

We first profile the runtime for each operation in Table 1 in the prefill and decode phase for various batch sizes and sequence lengths for the GPT-3 model [25]. We further profile the network communication cost to faithfully simulate tensor-parallel and pipeline-parallel executions. Finally, we build a regression model to extrapolate and predict these values for missing data points that may be encountered during an online simulated inference serving system. We confirmed that the estimated runtimes by the simulator are within 5% of the empirical values on an 8-GPU, 80GB A100 DGX box.

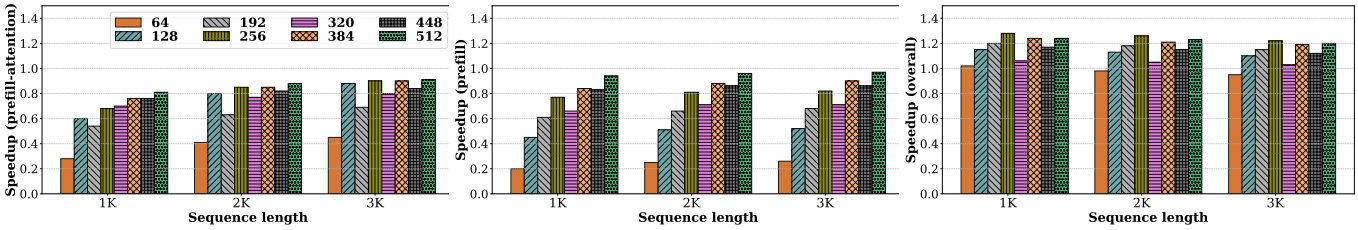
We report results for deployment over 64 A100 GPUs across eight servers connected with InfiniBand. We evaluate three scenarios; (1) 8-way tensor-parallel (TP) within a node with 8-way pipeline-parallel (PP) across nodes with the best-case Orca-style scheduling, (2) the same TP-PP setup as above with scheduling using SARATHI's *chunked-prefills* and *decode-maximal batching*, (3) 8 parallel replicas, each with 8-way TP, serving simultaneously. For all scenarios, we use the maximum batch size that fits the GPU — for TP+PP this was 27 and for TP only this was 11. The P:D ratio is fixed at 10 for this simulation with the minimum and maximum sequence length of the requests set to 1K and 4K respectively. Each request may have a different sequence length which is sampled from a Zipf distribution ($\theta = 0.4$), adhering to the maximum sequence length. The number of prefill and decode tokens is then calculated by satisfying the desired P:D ratio. For this experiment, we set the chunk size to be 256.

Figure 12a plots the cdf of pipeline bubble time per request. We define this as the sum of bubble time for all the micro-batches across all iterations for a given request. SARATHI reduces the median bubble time per request by 6.29×, by creating equal-compute units of work.

Next, we compare the overall request completion time for the different scenarios in Figure 12b. This graph plots the time to complete a given number of requests (our simulation considers a total of 10K requests). The TP-PP execution requires less memory for storing parameters compared to the TP-only setup, resulting in more room for the KV cache. Thus the TP-PP deployment supports 2.45× higher batch size compared to TP-only deployment, and yet, we observe that the TP-only execution is 1.28× faster than the baseline TP-PP with Orca scheduling, due to the large pipeline bubbles in the latter case. However, with *chunked-prefills* and *decode-maximal batching*, SARATHI enabled PP execution is accelerated by 1.91× compared to the baseline TP-PP, and by 1.48× compared to the TP-only execution. Thus, SARATHI makes pipeline parallel execution an attractive option for LLM inference by significantly minimizing pipeline bubbles.

5.4 Ablation Study of *Chunked-prefills*

In this subsection, we evaluate how splitting a full prefill computation into multiple smaller prefill chunks affects the efficiency of the prefill stage in SARATHI. To quantify this, we



(a) Self-attention (prefill-only). (b) chunked-prefills vs. full prefill. (c) End-to-end speedup for the entire batch.

Figure 13: **Ablation study:** Effect of varying the chunk size on different components of the system for LLaMa 13B on A6000 GPU.

measure the time to compute the prefill phase for various sequence lengths using the full sequence at once - this represents our baseline prefill performance. For each long sequence, we then compute the prefill with *chunked-prefills* and compare its end-to-end runtime with the baseline. The difference between the two indicates the overhead of *chunked-prefills*.

Prefill chunking has two potential sources of overheads: (1) it uses smaller chunk sizes compared to the baseline which may lower the GPU utilization, and (2) it needs to load the KV cache of each chunk multiple times, depending on the number of chunks in a request. Therefore, to fully understand the overhead of prefill chunking, we investigate the following: (1) what is the impact of chunking on attention computation for a prefill-only batch, (2) what is the effect of chunking on the overall runtime of prefill-only batch, and (3) what is the end-to-end throughput when *chunked-prefills* is used in tandem with *decode-maximal batching*. We study these by varying the chunk size from 64 to 512 as shown in Figure 13.

First, we observe that smaller chunk sizes can add significant overhead, for both attention and the overall prefill runtime. For example, the chunk size of 64 incurs 3× overhead for attention (see Figure 13a) and about 5×(see Figure 13b) in the overall prefill time. As one can expect, the overhead of *chunked-prefills* is lower for large chunk sizes: this is a combined effect of higher GPU utilization and fewer KV cache reloads with larger chunks. Overall, we find that chunk sizes of 256 and 512 provide reasonable prefill efficiency, limiting the end-to-end prefill computation loss to within 20% and 10%, respectively.

Second, SARATHI can compensate for some loss in prefill efficiency by improving the decode throughput. For instance, we see from Figure 13c that a chunk size of 64 almost matches the performance of our baseline despite being 5× slower in prefill whereas a chunk size of 128 yields up to 1.16× higher throughput despite its prefill being more than 2× slower than the baseline, mainly due to piggybacking more decodes. The tile-quantization effect is also evident in Figure 13 as SARATHI achieves higher improvement in throughput when the chunk size is a multiple of 128; e.g., chunk size 256 shows better speedup than 320.

6 Discussion

In this paper, we have comprehensively demonstrated how SARATHI improves the performance of LLM inference across several models and hardware configurations. However, there are multiple challenges that require further investigation.

First, we focus only on an efficient scheduling mechanism in SARATHI to improve the throughput of LLM inference. However, real-world deployments need to optimize an inference serving infrastructure simultaneously along multiple dimensions e.g., latency, queuing delays, fairness, etc. Meeting these goals with SARATHI requires revisiting scheduling policies. Second, although we show what is an appropriate chunk size for a given P:D ratio, we leave it to future work to explore how to pick an optimal chunk size as it depends on several factors like the hardware, model characteristics, sequence length, and the composition of prefill-decode tokens, especially in scenarios where the P:D ratio may not be known ahead of time. Third, we make a simplistic assumption in this paper that each request in a batch has the same number of prefill and decode tokens (except the simulation experiments) whereas, in the real world, the sequence lengths can vary significantly across different LLM inference requests. Finally, we focused on sequence lengths of up to 3K, and P:D ratio in the range of 1-200. We believe that these are representative of many real-world deployments. However, there has also been an increased interest in supporting very long sequences (e.g., 10s-100s of thousands [18]). Such large sequence lengths may pose new challenges as the cost of attention grows quadratically with the number of tokens. We are actively investigating these challenges.

7 Related Work

In this section, we provide a brief summary of related work along two dimensions: systems optimizations and model innovations.

7.1 Systems Optimizations

Memory management: In auto-regressive decoding, the number of tokens that need to be generated for a given request is not known apriori. Therefore, conventional systems pre-allocate memory for the KV cache based on a conservative estimation of the maximum number of tokens. Recently, vLLM showed that this approach is inefficient and proposed a framework — motivated by the virtual memory abstraction — that enables incremental memory allocation for KV caches [20]. This helps improve the batch size, especially when the number of tokens varies significantly across different requests. FlexGen [42] focuses on improving the throughput of offline LLM inference in resource-constrained scenarios e.g., running a large model on a single GPU. Toward this goal, FlexGen employs a judicious combination of memory offloading, quantization, and scheduling.

Optimizing (self-)attention: In [40], the authors propose an algorithm to reduce the memory requirement of self-attention from $O(n^2)$ to $O(1)$, with respect to the sequence length. FlashAttention [29] proposed a tiling-based algorithm that speeds up attention computation by minimizing the number of bytes read/written between different levels of GPU memory. Follow-up work [28] on FlashAttention further improved it along parallelism and work partitioning [28]. In our experiments, we found the xformers memory efficient attention implementation [21] to be the most efficient.

Kernel-level optimizations: FasterTransformer [6] proposed optimized layers for the transformer's encoder and decoder blocks. These are based on low-level GPU optimizations such as kernel fusion. We expect that such low-level optimizations would equally benefit SARATHI as well.

Scheduling optimizations: Orca proposed an iteration-level scheduling framework that avoids wasting compute due to token padding that was used earlier to batch together requests with different sequence lengths [48]. Further, Orca reduces latency by returning the response as soon as a request's end-of-sequence token gets generated. FastServe proposed a preemptive scheduling framework to minimize the job completion times [46]. Some other scheduling frameworks include Triton [13] and Clipper [27] that separate the serving layer from the execution engine of the model. Our current work focuses on optimizing the execution layer and can be used with different scheduling policies proposed by such systems.

The optimizations proposed by several of the prior works can complement our optimizations e.g., more optimized attention implementations will enable scaling SARATHI to longer sequence lengths and dynamic memory allocation will help in supporting larger batch sizes and so on.

7.2 Model Innovations

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 shares the same keys and values across all the attention heads to reduce the size of the KV cache [41], allowing larger batch sizes. Several recent works have also shown that the model sizes can be compressed significantly using quantization [30–32, 47]. Mixture-of-expert models are aimed primarily at reducing the number of model parameters that get activated in an iteration [23, 33, 36]. More recently, retentive networks have been proposed as a successor to transformers [44]. In this work, we focus on addressing the performance issues of the most popular transformer models from a GPU's perspective. Model innovations are orthogonal to our work.

8 Conclusion

In this paper, we identify two primary reasons for LLM inference inefficiency: 1) suboptimal GPU utilization due to lack of parallelism and memory-bound nature of decode phase, and 2) significant pipeline bubbles due to inconsistent prefill and decode times across different iterations, leading to micro-batch imbalance. To address these challenges, we introduce SARATHI, a novel approach that incorporates *chunked-prefills* and *decode-maximal batching*. *Decode-maximal batching* improves GPU utilization by piggybacking decodes with prefills, which converts the memory-bound decode phase to be compute bound. *Chunked-prefills* helps with making more prefills available for decodes to piggyback, and also provides for a uniform unit of work which helps significantly reduce pipeline bubbles. We demonstrate that SARATHI results in significant improvements in end-to-end throughput across models and hardware configurations.

References

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

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

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

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

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

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

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

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

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

[10] Llama model card. https://huggingface.co/decapoda-research/llama-13b-hf.

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

[12] nanogpt. https://github.com/karpathy/nanoGPT.

[13] NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.

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

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

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

[17] Text generation inference. https://huggingface.co/text-generation-inference.

[18] The Secret Sauce behind 100K context window in LLMs: all tricks in one place. https://blog.gopenai.com/how-to-speed-up-llms-and-use-100k-context-window-all-tricks-in-one-place-ffd40577b4c.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

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

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.