Medha: Efficiently Serving Multi-Million Context Length LLM Inference Requests Without Approximations

Amey Agrawal[2] Haoran Qiu[1] Junda Chen[3] Íñigo Goiri[1] Chaojie Zhang[1] Rayyan Shahid[2]
Ramchandran Ramjee[1] Alexey Tumanov[2] Esha Choukse[1]
[1]Microsoft [2]Georgia Institute of Technology [3]UC San Diego

Abstract

As large language models (LLMs) handle increasingly longer contexts, serving long inference requests of millions of tokens presents unique challenges. We show that existing work for long context inference is largely based on techniques from long context training, and does not handle the high variability in input lengths during inference. This leads to inefficient resource utilization, server fragmentation, and head-of-line (HOL) blocking.

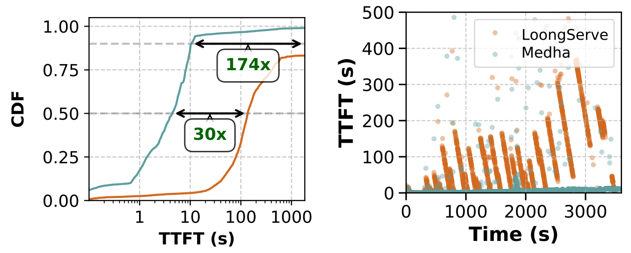
We present Medha, an end-to-end system for efficient long-context LLM inference that addresses these challenges through fine-grained time sharing. Medha introduces three key innovations: (1) the mechanism of adaptive prefill chunking to help mitigate HOL blocking with preemption; (2) two new parallelism strategies: Sequence Pipeline Parallelism (SPP) to reduce time-to-first-token by pipelining prefill chunks, and KV-Cache Parallelism (KVP) to lower time-per-output-token by distributing decoding across servers; and (3) a novel input-length aware least remaining slack scheduling to meet Service Level Objectives (SLOs).

Medha enables exact inference scaling beyond 10 million tokens, maintaining high throughput and low latency across mixed-length workloads. Compared to state-of-the-art systems, Medha reduces server fragmentation, cuts median latency by up to 30×, and improves throughput by over 5×, delivering production-scale long-context inference without compromising performance on shorter requests.

1 Introduction

Motivation. Emerging applications (*e.g.*, book summarization, movie analysis, multi-agent dialogue with knowledge retrieval, and multi-modal reasoning) are pushing large language models (LLMs) to process contexts spanning millions of tokens, orders of magnitude longer beyond the limits of current systems [14, 30, 51].

The quadratic cost of self-attention [47] makes long-context inference latency-intensive. Recent training-time methods like Context Parallelism (ring and striped attention [9, 28]) distribute context processing across a *fixed* number of GPUs to mitigate this cost. However, unlike training, inference requests have highly variable sequence lengths. Thus, using a fixed degree of parallelism leads to significant communication overhead and inefficient compute/memory utilization.



(a) TTFT distribution where LoongServe shows 30-174× higher latency due to its coarse-grained space-sharing.

(b) TTFT over time showing head-of-line blocking for LoongServe where short requests get stuck behind long requests.

Figure 1. Impact of long-context requests on TTFT for Llama-3 8B inference using 16 A100 GPUs with LoongServe [49] and Medha.

LoongServe [49] addresses variability in sequence lengths using elastic context parallelism and prefill-decode disaggregation, dynamically resizing GPU pools to match request lengths. While this reduces prefill latency and communication overhead, it remains vulnerable to head-of-line (HOL) blocking—an issue inherent to ring attention. As shown in Figure 1, LoongServe experiences sharp latency spikes when serving a mix of short and long-context requests (with just 5% long requests), as short ones are blocked behind long-running requests.

Moreover, LoongServe *fragments* servers into dedicated pools for prefill and decode stages, further reducing overall resource utilization, as we illustrate later in Figure 4. A common workaround is to introduce more pools partitioned by request length buckets. However, determining the right splits is challenging (*e.g.*, a 128K-token request can be 100× cheaper than a 1M-token one), often resulting in inefficiency and even greater server fragmentation.

Our Work. We design Medha, a system for scalable long-context inference that reduces resource fragmentation and improves efficiency across all request types (*i.e.*, prefill and decode for both short and long requests). To achieve this, Medha enables fine-grained preemption, and uses advanced scheduling techniques to share the resources better, avoiding the inefficiencies of static partitioning.

The key enabling mechanism in Medha is *prefill chunking*, which splits long-context requests into smaller chunks, allowing prefill and decode to be batched together. This line of work tries to utilize resources better without creating pools. Contrary to prior belief [4, 17], chunking overhead decreases with context length; even small chunks (*e.g.*, 64 tokens) add minimal cost at scale (Figure 5b). We leverage this insight by introducing, *adaptive chunking*, wherein we dynamically adjust the chunk size for a request based on the context length to meet decode latency SLO without compromising on system throughput (Figure 6b). This fine-grained preemption ability also allows short requests to interleave with long ones and complete without delay, using our slack-aware prioritization across prefill requests.

However, chunking alone is insufficient since ring attention is incompatible with chunked prefills, and tensor parallelism (TP) cannot scale across servers due to interconnect limitations. To overcome this, Medha introduces two new parallelism strategies: (1) *Sequence Pipeline Parallelism (SPP)* reduces time-to-first-token (TTFT) by concurrently processing consecutive chunks of a long request across pipleine stages, enabling linear scaling of prefill latency; and (2) *KV-Cache Parallelism (KVP)* accelerates token generation by distributing the KV-cache across servers, reducing time-per-output-token (TPOT).

Medha unifies TP, SPP, and KVP in a novel 3D parallelism framework and combines it with adaptive chunking and batching. This design eliminates fragmentation, avoids HOL blocking, and enables exact inference with context lengths up to 10 million tokens. In this way, Medha enables exact inference with long contexts, achieving performance scaling for context lengths up to 10 million tokens. Medha reduces median latency by up to 30× and improves throughput by over 5× compared to state-of-the-art systems.

Summary. We make the following contributions to long-context inference serving systems, *without approximations*:

- *Sequence pipeline parallelism (SPP)* for concurrent chunk execution during prefill, reducing TTFT while mitigating HOL blocking.
- *Adaptive chunking* and *KV cache parallelism (KVP)* to dynamically control the trade-off between TTFT and TPOT across requests in mixed-batching scenarios with long context requests.
- *Medha 3D parallelism* combining TP, SPP, and KVP to scale inference to 10M tokens, maintaining efficiency and low latency across heterogeneous batches.
- *Medha scheduling* that prioritizes and balances short and long requests using fine-grained time sharing.

2 Background and Motivation

2.1 Long-Context LLM Inference

Models. Recent research showed that LLMs can be fine-tuned to handle context lengths spanning millions of tokens by re-scaling positional embeddings [8, 27, 46, 51]. These long-context transformers unlock new capabilities, including deep agentic workflows, multi-modal processing and reasoning over several books' worth of textual data. For example, Google's Gemini 1.5 model [40] supports contexts of up to 2 million tokens in production, and Meta's Llama 4 supports up to 10 million context length [30].

Table 1. Definitions of notations in equations.

Notation	Definition
n	number of tokens
h_q or h_{kv}	number of query or key-value heads
d	attention head dimension
p_j	parallelism degree for strategy j. *e.g.* p_{tp} for TP
M_{kv}	memory required for KV cache
F_a	attention flops
R_a	number of bytes read for attention
I_a	attention arithmetic intensity
c	chunk size
T	execution time
T_p or T_d	prefill latency or decode latency

Phases and Metrics. LLM inference with auto-regressive transformers involves two distinct phases, each with unique resource and performance profiles [5, 35]. The prefill phase is compute-intensive, processing input tokens and building the KV cache. Its latency defines Time to First Token (TTFT), which is critical for interactive use. The decode phase generates output tokens sequentially, dominated by memory bandwidth due to frequent KV cache reads. The Time per Output Token (TPOT) affects the model's fluency.

Resource Requirements. With longer contexts, computational complexity grows quadratically with input size. For example, serving 1 million tokens using Llama-3 70B requires 320 GB of memory for the KV cache and 2.4 exaFLOPs.

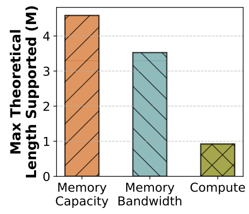
As we present our analysis of these resources, Table 1 summarizes our notation. During prefill, each token attends to all prior tokens, involving two matrix multiplications: (1) query (Q) and key (K) tensors to obtain the attention matrix and (2) attention matrix with value (V) tensors. Each operation requires $2n^2dh_q$ FLOPs. In causal attention, only the lower triangular matrix is computed, halving the compute cost, but still growing quadratically with number of tokens. Thus, for n input tokens, the compute FLOPs are:

$$F_a(n) = 2n^2dh_q \tag{1}$$

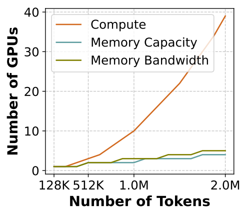
During decode, we scan the entire KV cache so far, resulting in a linear increase in memory reads. The capacity for the KV cache and memory reads are:

$$M_{kv}(n) = 4ndh_{kv} = R_a(n) \tag{2}$$

Figure 2a shows the theoretical maximum input tokens that meet a 30s TTFT and 20ms TPOT SLO on a single DGX-H100 node for Llama-3 8B. Compute becomes a bottleneck at ~1M tokens, and memory capacity at 4M tokens. Figure 2b



(a) Maximum number of tokens per resource type on 8 H100 GPUs.



(b) GPUs required to meet each resource for given context length.

Figure 2. Theoretical resource requirements for serving Llama-3 8B with 30s TTFT and 20ms TPOT SLOs. Compute is the primary scaling bottleneck for interactive long-context LLM inference.

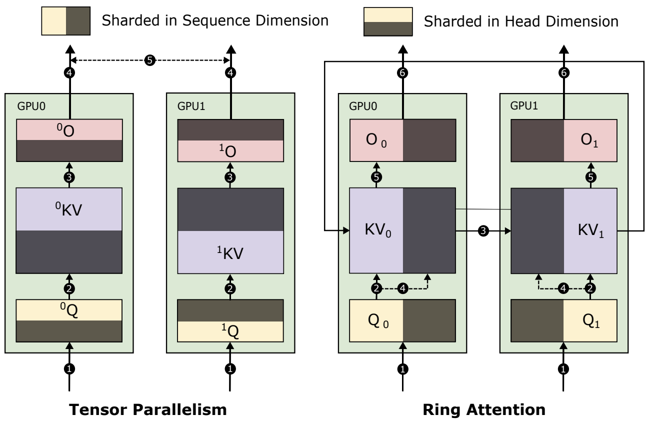


Figure 3. TP shards computation across the head dimension, Ring Attention (one CP implementation) distributes computation across the sequence dimension with cyclic KV cache transfers. Arrows show data flow, with numbered steps showing computation order.

shows the GPUs required to meet this SLO as input tokens increase: 10 GPUs for 1M tokens and 40 for 2M.

2.2 Serving Long Context Queries

To serve modern LLMs with billions of parameters, long-context requires distributed computation across multiple GPUs for high throughput and interactive latencies.

Traditional Parallelism Techniques. *Tensor Parallelism (TP)* [43] divides tensors within layers, distributing operations like attention across devices (Figure 3). This intra-layer strategy improves both latency and throughput, but frequent large-scale communication limits its scalability. TP typically requires high-speed interconnects (*e.g.*, NVLINK), confining it to single-server deployments on systems like DGX-H100.

Pipeline Parallelism (PP) [5, 20, 54] distributes model layers across devices, easing memory pressure and freeing up space for KV cache to support larger batches and higher throughput. While it scales well across nodes due to minimal inter-stage communication, it offers no latency benefit due to sequential stage dependencies.

As context lengths grow into the millions of tokens, achieving interactive latencies requires efficient parallelization across many devices. ***Traditional approaches fall short: TP does not scale well, while PP improves throughput but not latency.***

LoongServe. A recent work LoongServe [49] uses Ring Attention/Context Parallelism [9, 28] (CP) for long context serving (Figure 3), and therefore partitions queries across devices, with each shard computing attention over rotating blocks of KV cache. When each device processes enough tokens (*e.g.*, 24.5K on A100 with InfiniBand [28]), communication overlaps with computation, allowing efficient multi-GPU scaling for long-context workloads. However, this requires LoongServe to dynamically adjust the degree of parallelism based on input length and manage separate resource pools. Large requests are distributed across more GPUs for parallel execution, while smaller ones are packed efficiently to maximize utilization.

Furthermore, since CP is ineffective for decoding, LoongServe follows prior work [35, 37] in using disaggregation, migrating the KV cache to different devices for decoding. As shown in Figure 2b, decode is memory-bound and uses fewer devices than the compute-bound prefill phase. Existing systems built on CP [52] are subsumed by LoongServe in both capability and performance.

2.3 Limitations of LoongServe

Our analysis shows that LoongServe faces fundamental limitations due to its coarse-grained resource management.

Resource Fragmentation. LoongServe partitions cluster resources into three types of isolated pools for: short prefills, long prefills, and decodes. This segmentation limits flexibility by forcing the same ratios on memory and compute across prefill and decode: long requests cannot fully utilize available compute, and resources remain underused within artificial boundaries. Figure 4 shows this effect from fragmentation, where at times, with very long requests, the compute demand for prefill shoots up dramatically due to the quadratically cost of attention. ***The compute in decode pools is often underutilized due to memory-bound workloads, while long-prefill requests are slowed down due to reduced compute capacity due to fragmentation.***

Head-of-Line Blocking. The biggest limitation of LoongServe is its poor handling of mixed workloads. Long-context requests (*e.g.*, 1M tokens) occupy majority of resources for minutes during prefill, causing severe head-of-line blocking for other requests. Note that, while LoongServe attempts to form multiple prefill pools to tackle this challenges, the large variance in prefill computation cost with sequence length inevitably results in HOL blocking for some requests. ***Though LoongServe allows space sharing, its coarse-grained elastic allocation and lack of preemption limit effective resource multiplexing.***

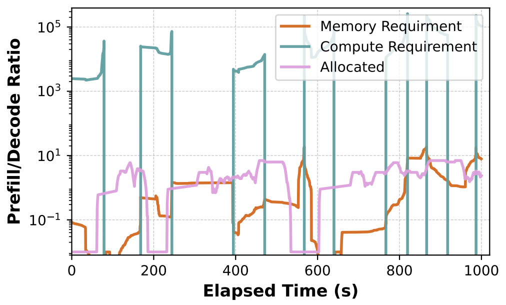


Figure 4. Timeline showing that as requests arrive and drain, the required memory and compute ratios for prefill and decode diverge significantly (note the log-scale y-axis). Prefill pools become compute-bound, while decode pools are limited by memory capacity. LoongServe's static partitioning approach leads to fragmentation of memory and compute resources across pools, reducing its ability to efficiently serve such divergent resource demands.

Takeaway: *The coarse-grained space sharing in ECP is insufficient. For efficient long-context serving, we require a fundamentally different approach that combines fine-grained time- and space-sharing.*

3 Medha: Key Insights & Mechanisms

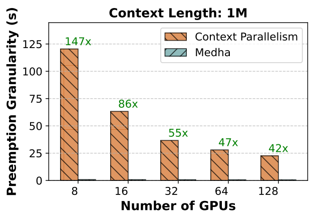
The ***main insight*** driving Medha is that for very high variability in context lengths, pool creation causes resource fragmentation. Therefore, we optimize techniques that run large prefills piecewise, enabling fine-grained preemption across a mix of workload ***on a shared set of hardware.*** The parallelism techniques we propose, unlike Context Parallelism or LoongServe, ***do not need rigid sizing*** proportional to context lengths. And finally, our batching and scheduling algorithms are tuned to ***offer SLO-awareness for a wide variety*** of context lengths.

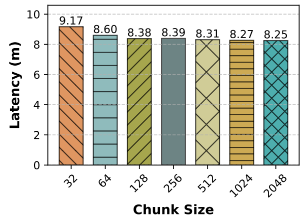
3.1 Chunked Prefills for Long Context

Instead of executing the entire prefill at once, chunking the input prompt during prefill [4] enables more flexible scheduling. This approach allows finer-grained control, improving adherence to both prefill and decode latency SLOs across varying context lengths. As shown in Figure 5a, finer preemption granularity leads to more efficient and responsive scheduling. This technique parallels the disaggregation of prefill and decode phases [35] employed by LoongServe.

The Myth. Chunked prefills cause read amplification, since the whole KV cache must be read for each chunk's attention, increasing KV cache reads from $O(n)$ to $O(n^2)$. This has led to the myth that chunked prefills are inefficient for long contexts [5, 49, 57]. We challenge this assumption through an analysis of arithmetic intensity.

Busting the Myth. We find that the arithmetic intensity of a prefill chunk depends solely on the chunk size, not the total context length. Each chunk processes c tokens, requiring reads of the full KV cache but performing a fixed number

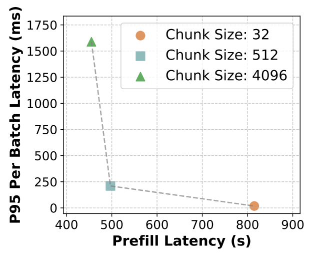


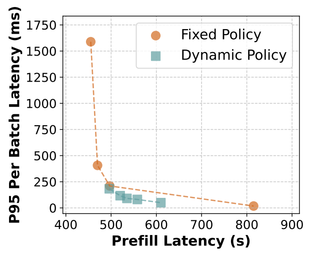


(a) Preemption granularity enabled on 1M token sequences prefill with Llama-3 8B.

(b) Self-Attention computation time with chunked prefill for 1M tokens with Llama-3 70B using 8 H100s.

Figure 5. Efficacy of chunked prefill for long-context inference.





(a) Static chunk sizes.

(b) Adaptive chunk size.

Figure 6. Pareto frontiers of prefill/decode latencies in mixed batching with chunked prefills: (a) Static sizes have a trade-off between prefill and decode latencies. (b) Adaptive chunking starts with larger chunks, gradually reducing size to keep batch latencies consistent, achieving better prefill efficiency and low decode latency.

of operations per token. So, while longer contexts increase memory reads, compute per chunk stays constant.

Modern LLMs amplify this effect via grouped-query attention [6] (Equation (3)). For example, in Llama-3 70B, 8 query heads share one KV head, boosting arithmetic intensity nearly 8× over standard linear layers [4]. As a result, on NVIDIA H100 GPUs running Llama-3 70B, a chunk of just 40 tokens can saturate GPU compute.

$$I^i_{cp}(n,c) = \frac{F^i_{cp}(n,c)}{R^i_{cp}(n,c)} \simeq \frac{4ic^2dh_q}{4icdh_{kv}} = c\frac{h_q}{h_{kv}} \tag{3}$$

This high intensity enables splitting multi-million-token prefills into thousands of short chunks. Each chunk runs in tens of milliseconds, making prefill fast and interruptible; unlike CP's long, monolithic executions which take minutes.

Kernel Bottlenecks. Early attention kernels struggled with chunked prefills, as they parallelized only over query (Q) tokens; limiting throughput when Q is small. Recent work addresses this with KV-parallel designs: FlashDecoding [18, 41] shards across KV tokens, while FlashInfer and FlashAttention-2 [11, 53] parallelize over both Q and KV, delivering efficient performance at any context length.

3.2 Adaptive Chunked Prefills

Aligned with our analytical model, Figure 5b shows that a chunk size of 32 adds just 11% overhead to self-attention

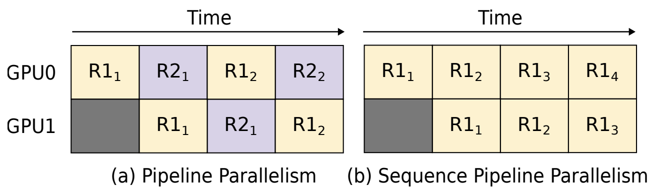


Figure 7. Contrasting PP strategies for prefill processing. (a) Standard PP interleaves micro-batches composed of prefills from different requests ($R1$, $R2$) to improve throughput. (b) SPP overlaps chunks of the same request ($R1_1$, $R2_2$) across stages to reduce prefill latency for long request while maintaining high GPU utilization.

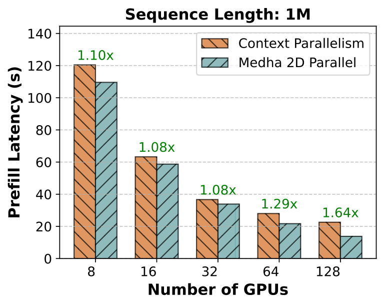


Figure 8. Performance comparison of Context Parallelism vs. Medha 2D Parallel (SPP+TP) for 1M token sequences prefill with Llama-3 8B. Medha achieves better scaling efficiency for prefill computation, resulting in up to 1.64× lower prefill latency.

compared to a 2048-token chunk. However, small chunks can hurt end-to-end performance due to inefficient linear layer execution and fixed CPU overheads. For Llama-3 8B on 8 NVIDIA H100 GPUs, a 32-token chunk increases prefill latency by 1.75× for a 1M-token request compared to 4096-token chunks. Larger chunks, in contrast, increase decode latency for batched requests (Figure 6a), creating a trade-off between prefill and decode efficiency.

Early in prefill, MLP layers dominate runtime; as the KV cache grows, attention becomes the bottleneck, where smaller chunks become more efficient. To balance this, we begin with large chunks and shrink them over time.

Our adaptive chunking strategy dynamically adjusts chunk size during prefill based on (a) decode latency SLOs of other batched requests, and (b) the minimum efficient chunk size at the current stage. This approach significantly improves the prefill–decode latency trade-off, as shown in Figure 6b.

3.3 Sequence Pipeline Parallelism (SPP)

While chunked prefills avoid HOL blocking, we still need efficient parallelization to reduce latency for long-context requests. As noted in Section 2.2, TP does not scale well due to high communication overhead. PP, used in systems like Orca and Sarathi-Serve [4, 54], maintains efficiency by interleaving micro-batches from different requests (Figure 7).

This works for decode, where outputs have strict sequential dependencies. However, it is suboptimal for prefill. We observe that the prefill chunks do not rely on the model output from the previous chunk.

Table 2. Comparison of parallelization techniques for long-context LLM inference. *Preemption indicates if it supports fine-grained preemption with chunked prefills.

Parallelism strategy	Batching	Preemption*	Fast prefill	Fast decode	Scalable
Pipeline Parallelism (PP) [20]	✓	✓	×	×	↑
Tensor Parallelism (TP) [43]	✓	✓	✓	✓	↓
Context Parallelism (CP) [9, 28]	×	×	✓	×	↑
Sequence Pipeline Parallelism (SPP)	✓	✓	✓	×	↑
KV Parallelism (KVP)	✓	✓	✓	✓	↓
Medha 3D Parallelism (3DP)	✓	✓	✓	✓	↑

This motivates the design of Sequence Pipeline Parallelism (SPP): a novel pipelining strategy that cuts prefill latency through smarter chunk scheduling. The core idea lies in scheduling chunk $i + 1$ right after chunk i finishes the first pipeline stage (Figure 7). This tightly packed schedule maximizes pipeline utilization during prefill, enabling near-linear speedups as GPU count scales:

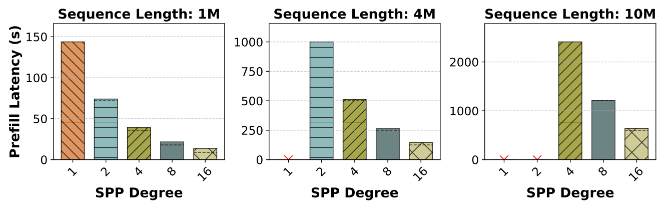
$$T_p^{spp}(n,c) \simeq \frac{T_p(n,c)}{p_{spp}} + \frac{T_{comm}^{pp}(c)n}{c} \sim \frac{T_p(n,c)}{p_{spp}} \tag{4}$$

$T_p^{spp}(n,c)$ is the SPP prefill time for n tokens with chunk size c, $T_p(n,c)$ is the standard prefill time, p_{spp} is the SPP degree, and $T_{comm}^{pp}(c)$ is the inter-stage communication time. The communication overhead $\frac{T_{comm}^{pp}(c)n}{c}$ becomes relatively negligible as n increases due the quadratic scaling of $T_p(n,c)$, enabling communication-efficient cross-node scaling.

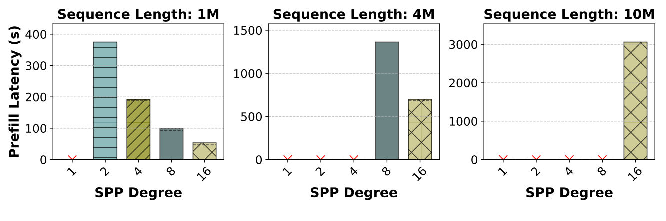
In addition to supporting batching and preemption, this approach presents a distinctive advantage over context parallelism: the effectiveness of SPP remains independent of variations in input sequence length, unlike CP, where the degree of parallelism is closely tied to the sequence length.

Faster TTFT with Medha 2D (SPP+TP). Figure 8 compares the prefill latency of CP [9] (the best baseline for long-context prefill) with Medha 2D SPP+TP. Medha achieves 64% lower latency than CP using 128 H100 GPUs (16 servers) when processing one million tokens, with TTFT latency under 15 seconds while using a 4K chunk size.

Scaling SPP to 10M Tokens. Figure 9 shows the TTFT of Medha 2D SPP+TP as the token count increases from 1M to 10M, with pipeline depth for SPP varied from 1 to 16 for Llama-3 8B and Llama-3 70B. Red crosses indicate configurations that are infeasible due to memory limitations. Medha 2D SPP+TP scales nearly linearly with pipeline depth, benefiting from the optimizations in Section 4.4. The strong scaling trendlines suggest that more servers could further reduce TTFT for larger contexts.

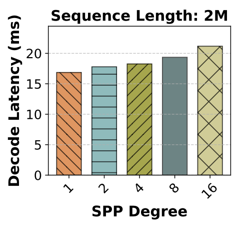


(a) Llama-3 8B.

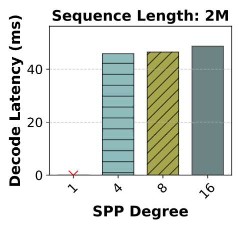


(b) Llama-3 70B.

Figure 9. Scaling efficiency of Medha 2D (SPP+TP) for long-context prefill processing. Medha 2D reduces TTFT near-linearly (80%+ scaling efficiency) as the SPP degree increases to operate with up to 128 H100 GPUs. Red crosses are infeasible settings due to memory limitations.

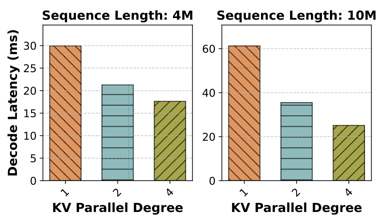


(a) Llama-3 8B

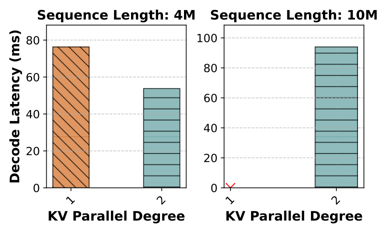


(b) Llama-3 70B

Figure 10. Impact of SPP scaling on decode latency in Medha 2D (SPP+TP, $p_{tp} = 8$). Decode latency is only marginally affected even with a 16-stage pipeline.



(a) Llama-3 8B with $p_{spp} = 4$.



(b) Llama-3 70B with $p_{spp} = 8$.

Figure 11. TPOT reduction with KVP in Medha 3D in decode-only batches. For 10M context length decodes for Llama-3 8B, $p_{kvp} = 2$ results in almost 40% reduction in latency, allowing decode at the rate of ~30 tokens per second.

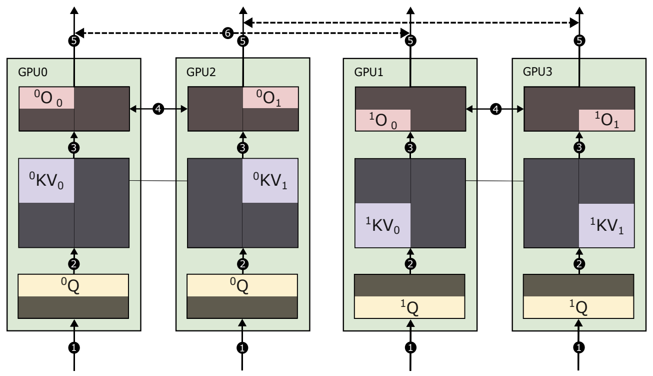


Figure 12. Combining KVP (horizontal) and TP (vertical) in Medha. KVP splits the sequence across GPUs 0-1 and 2-3 while TP divides computation within each KV shard across attention heads.

Decode Latency Impact of SPP. Having shown how SPP unlocks lower TTFT, we now show in Figure 10 the decode latency achieved as SPP scales out for 2M context on Llama-3 8B and Llama-3 70B. Given the small overheads of PP, decode latency has minimal impact as we scale SPP degree p_{spp}. Note that the decode latency impact of SPP is further minimized as we move to larger model (e.g., Llama-3 8B to Llama-3 70B). This is caused due to the similar communication overhead but higher computation time per stage in large models.

3.4 KV Cache Parallelism (KVP)

While SPP offers an effective mechanism to reduce prefill latency, it does not improve decode latency due to the cross-iteration dependency in auto-regressive decoding. To address this challenge, we propose KV Cache Parallelism (KVP), a novel technique that effectively reduces decode latency by parallelizing KV-cache reads.

KVP shards the KV cache across multiple GPUs along the sequence dimension as shown in Figure 12. During each iteration, we replicate the Q token(s) across all GPUs and compute partial attention outputs based on each local KV-cache shard. These partial outputs are then combined using online-softmax [33]. A critical advantage of KVP over techniques like Ring Attention is that the communication cost T^{kvp}_{comm} is independent of the KV-cache length and only depends on the number of query tokens. This makes KVP extremely effective in managing decode latency for long-context requests. The KVP performance improvement can be modeled as:

$$T^{kvp}_d(n) \simeq \frac{T^{attn}_d(n)}{p_{kvp}} + (T_d(n) - T^{attn}_d(n)) + T^{kvp}_{comm} \quad (5)$$

Where $T^{kvp}_d(n)$ is KVP decode time, $T^{attn}_d(n)$ is the attention computation time, p_{kvp} is the KVP degree, $T_d(n)$ is the total decode time, and T^{kvp}_{comm} is the communication overhead.

KVP extends naturally to chunked prefills. For long sequences, the communication cost of KVP (${}^{i}T^{kvp}_{comm}(c)$) becomes significantly smaller than the attention computation

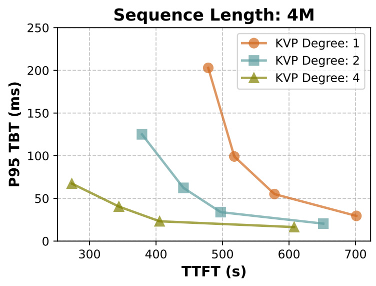


Figure 13. TTFT vs. TPOT trade-off for Llama-3 8B using Medha 3D (TP-4, SPP-4) varying KVP degree and chunk size (32–256).

itself. This relationship can be expressed as:

$$^{i}T_p^{kvp}(n,c) \simeq \frac{^{i}T_p^{attn}(n,c)}{p_{kvp}} + (^{i}T_p(n,c) - {}^{i}T_p^{attn}(n,c)) + {}^{i}T_{comm}^{kvp}(c) \tag{6}$$

Where $^{i}T_p^{kvp}(n,c)$ is the prefill time for the i-th chunk with KVP, $^{i}T_p^{attn}(n,c)$ is the attention computation time for the chunk, $^{i}T_p(n,c)$ is the total prefill time for the chunk, and $^{i}T_{comm}^{kvp}(c)$ is the communication overhead for the chunk.

Our experiments shows that KVP is also effective in *capping the latency impact of prefills* on the decodes of other batched requests in mixed batching scenarios. For instance, when processing a 4 million context length request, the P95 decode latency (for requests batched along) with even a small chunk of 128 tokens reaches almost 100ms (Figure 13).

To optimize resource usage, we use a dynamic KVP worker allocation strategy. Each request starts with one worker, and workers are added once we exceed the worker KV-cache token limit. This enables KVP replicas to batch short requests independently while cooperatively handling long ones, ensuring efficient resource utilization across workloads.

3.5 Medha 3D Parallelism

To meet the demanding prefill and decode latency requirements in long-context LLM inference, Medha introduces a novel 3D parallelism (Figure 14), combining SPP, KVP, and TP to scale performance across hundreds of GPUs. SPP accelerates prefill, while KVP and TP enhance both phases.

Faster TPOT with Medha 3D parallelism. Figure 11 shows the TPOT for Llama-3 8B and Llama-3 70B with 4M and 10M context length in Medha 3D parallel, where $p_{tp} = 8$, $p_{spp} = 4$ for Llama-3 8B, $p_{spp} = 8$ for Llama-3 70B, and p_{kvp} is varied. $p_{spp} = 8$ was used for Llama-3 70B, as longer context lengths do not fit within $p_{spp} = 4$ (see Figure 9b).

Figure 11 shows that increasing p_{kvp} brings down the TPOT considerably, helping achieve interactivity targets. The latency benefit is not linear due to Amdahl's law, but gets more pronounced with longer context length. Increasing p_{kvp} from 1 to 4, therefore using 4× the GPUs For Llama-3 8B, reduces TPOT by only 1.7× for 4M context length, whereas for 10M context length this benefit increases to 2.5×. Therefore, we use KVP only for longer context lengths.

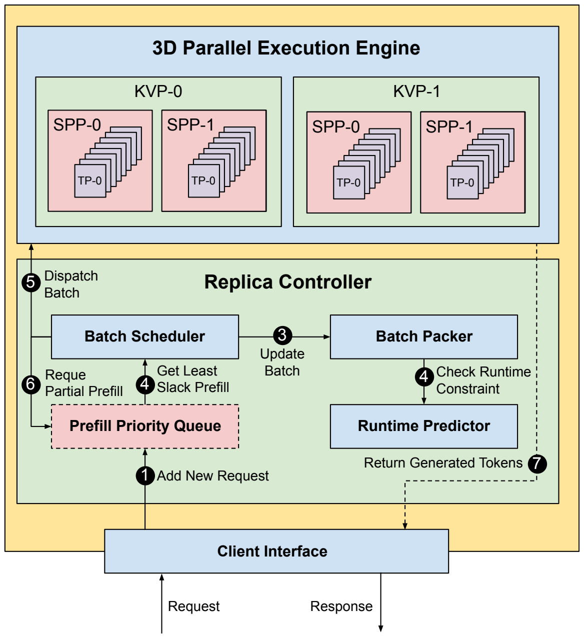


Figure 14. Medha architecture for efficient long-context inference. The *Replica Controller* centrally manages the request life-cycle, featuring a slack-aware *Batch Scheduler* and a *Batch Packer* to optimize for SLOs. It dispatches batches to the *3D Parallel Execution Engine*, which leverages Medha's novel combination of KVP, SPP, and TP.

Tackling Prefill-Decode Latency Tradeoff. We analyze how system parameters impact the tradeoff between prefill and decode performance. Figure 13 shows this tradeoff by examining the relationship between chunk sizes and KV cache parallelism (KVP) degree for Llama-3 8B with a 4M context length, while fixing SPP degree at 4. Increasing chunk size reduces TTFT (prefill latency) but increases TPOT. For a fixed chunk size, increasing p_{kvp} reduces both TTFT and TPOT in most cases, improving the overall tradeoff.

4 Medha: System Design and Implementation

Serving extremely long-context LLM requests requires balancing latency, efficiency, and resource fairness. Section 3 introduced mechanisms like adaptive chunked prefill, SPP, and KVP for long-context serving. Medha integrates these into a complete system with advanced scheduling, batching, and load-balancing to meet latency SLOs and optimize resource use. The design goals of Medha include:

(R1) Meet the TTFT and TPOT latency SLOs of both long and short context interactive requests.

(R2) Drive up the hardware utilization to increase throughput per device, thereby reducing operation cost.

(R3) Avoid HOL and provide fairness to efficiently handle mixed requests with a wide range of context lengths simultaneously within a single serving system.

4.1 Medha Architecture Overview

Medha employs a carefully designed architecture to efficiently serve mixed long and short context LLM requests, combining SLO-aware scheduling with 3D parallel execution. Figure 14 provides a visual representation of this architecture and the typical flow of a request.

Incoming requests first arrive via the **Client Interface** and are added to the **Prefill Priority Queue** within the **Replica Controller** (Step 1). The **Replica Controller** acts as the central coordinator, managing the request lifecycle to meet SLOs and maximize resource utilization.

The **Batch Scheduler** periodically queries the priority queue, selecting requests for the next batch based on a slack-aware policy (Step 4). This prioritizes latency-sensitive short requests and enables fine-grained preemption. The chunks sizes for the selected prefill requests is adjusted by **Batch Packer** (Step 3) to form executable batches with the help of **Runtime Predictor** to ensure SLO adherence (Step 4).

Batches are then dispatched by the **Replica Controller** (Step 5) to the **3D Parallel Execution Engine**. This allows Medha to scale across multiple servers to meet the TTFT and TPOT SLOs of long context serving, as discussed in Section 3.5 and compared in Table 2. The Batch Scheduler re-queues the partially completed prefill back into the waiting queue (Step 6), this enables us to use pick the highest priority prrefill across both new and partially completed requests.

4.2 Medha Scheduling and Batching Policy

The core of Medha's batching scheduler leverages slack-aware prioritization with fine-grained preemption that avoids HOL while meeting latency SLOs. Its design incorporates: (1) HOL avoidance by preventing long requests from blocking short ones; (2) fairness by fine-grained time-sharing among requests of varying context lengths; and (3) latency SLO compliance by meeting TTFT and TPOT deadlines.

All scheduling and batching optimizations in Medha are applied only during the prefill phase. Once a request enters the decode phase, it is never interrupted or preempted. This avoids violating tight TPOT SLOs and prevents wasting the KV cache memory already allocated for that request. To better handle compute variability from different request lengths, Medha extends the standard Least Remaining Slack (LRS) policy into an input-length-aware version, referred to as ILRS. Requests are prioritized dynamically based on their real-time slack, defined as the remaining fraction of time before their TTFT deadline is violated. This ensures that latency-sensitive requests, typically shorter ones, are scheduled first and not blocked by longer requests. To form each execution batch, Medha applies dynamic bin-packing heuristics that take into account both GPU resource constraints (compute and memory) and request deadlines.

The batching process continuously adjusts as new requests arrive, dynamically recomputing the batch composition at every iteration to maximize GPU utilization and meet deadlines, thereby preventing HOL blocking.

Slack-based Request Prefill Prioritization. The scheduler dynamically reorders the request queue based on ILRS (relative to the prefill deadline):

$$relative\_slack = \frac{request\_deadline - remaining\_prefill\_time}{request\_deadline\_duration} \quad (7)$$

Deadline latency is predicted at by the Runtime Predictor using offline profiling results collected and trained on Vidur [3], under the same system setup with isolated runs of requests across varied context lengths. By prioritizing requests with tighter deadlines, Medha reduces the chance of SLO violations. Because slack is defined as a percentage relative to each request's deadline, it avoids a common issue in Earliest Deadline First (EDF) scheduling, where long-context requests can be delayed by a steady stream of short arrivals.

Prefill-Prefill Batching (PPB). Slack-based prefill prioritization ends up favouring shorter prefills over longer prefills. However, we see an opportunity for almost free prefill-prefill batching. Consider a batch with early chunks of long-context request using a small chunk size (32–64 tokens). In this case, MLP layers are memory bound, underutilizing the compute capacity. Packing a prefill chunk from a short request leverages this arithmetic slack with minimal overhead.

Inspired by prefill-decode mixed batching, in Medha, we allow long prefill requests to share resources with short ones, enabling prefill-prefill batching (PPB). The fraction of space allocated to short requests in a mixed batch is proportional to their remaining slack. Requests closer to their deadline receive more resources, ensuring latency requirements are met. The slack fraction is capped by a configurable maximum sharing limit to prevent excessive overhead for long requests.

Batch Packing. The scheduler computes the number of tokens for the next batch chunk dynamically using the configured target batch time (*i.e.*, decode latency SLO) and the remaining slack fraction using dynamic-bin packing guided by runtime predictions. This ensures that, during prefill-decode mixed batching, the decode requests experience minimal delay the long-request prefill. This allows us to naturally integrate both adaptive chunking and prefill-prefill batching to improve prefill efficiency and reduce decode latency (Figure 6).

4.3 Medha KVP Load Balancer

Medha uses a just-in-time load balancing mechanism to efficiently allocate short and long-context requests to the appropriate compute resources. At the inter-KVP level, incoming requests are distributed based on the estimated runtime of pending prefill work and memory usage of each KVP rank, enabling efficient scaling across KVP dimension without causing resource hot spots. SPP complements this with implicit load balancing by evenly splitting prefill workloads along the pipeline stages.

4.4 Implementation Optimizations

Efficient long-context LLM inference requires platform-level optimizations. Medha extends the Sarathi-Serve framework [4] to tackle multi-million token context requests. Unlike vLLM and Sarathi-Serve, which incur overhead from centralized schedulers as sequence length grows, we reduce communication by replicating sequence state across the scheduler and GPU workers.

We replace Ray [12] with ZeroMQ [2] for scheduler-worker communication, eliminating GIL contention as we scale to hundreds of workers. We also integrate FlashInfer [53] kernels to distribute work across both query and KV tokens, optimizing chunked prefill for long contexts. To meet strict latency targets with small prefill chunks, we implement the model execution engine's critical path in C++ using PyBind, ensuring seamless integration with the Python codebase.

5 Evaluation

5.1 Evaluation Setup

Baselines. We compare our system against the state-of-the-art LLM inference serving systems, LoongServe [49] and vLLM [23] with long context request serving ability. Note that, for context lengths greater than 32K, vLLM defaults to the Sarathi-Serve scheduler [4]. Thus, we refer to this baseline as Sarathi. We consider two chunk sizes for the Sarathi scheduler: 512 and 2048. Furthermore, we also consider various disaggregated compute [19, 35, 57] options. However, none of the available open-source implementations support context lengths over 32K tokens, making them unsuitable for evaluation at the time of writing. Finally, we evaluate Medha variant that replaces the IRLS request prioritization and prefill-prefill batching with standard first-come-first-serve (FCFS) scheduling while retaining all other proposed mechanisms.

Models and datasets. We use Llama-3 8B and Llama-3 70B with RoPE [44] scaling to support up to 10M tokens. Currently, there are no publicly available long-context LLM datasets available that span millions of tokens. Previous systems use L-Eval [7] and LV-Eval [55] for long context evaluations. However, these datasets feature very short decode lengths (P90 < 75 tokens for LV-Eval), which do not represent real-world scenarios, where models must generate comprehensive responses after ingesting the context.

To evaluate realistic workloads, we construct the ***Medha-SWE*** trace using the Gemini-Flash-1.5B model [40]. We focus on two common software engineering tasks: code review and pull request (PR) handling. From the top 1,000 most-starred GitHub repositories with permissive licenses (Apache or MIT), we select those with token counts between 100K and 1M. We extract the 100 most recent issues and merged PRs per repo and prompt Gemini to respond to each, referencing the full codebase.

This yields interactions with prefill lengths of 393K (P50) and 839K (P90) tokens and decode lengths of 518 (P50) and 808 (P90). To simulate a realistic request mix, we combine these long-context examples with the ShareGPT4 trace [48], which consists of real GPT-4 conversations capped at 8K tokens. We test Medha under various ratios of long and short-context requests.

Hardware. We evaluate Medha across two hardware setups. For the Llama-3 8B model, we use a setup with two DGX-A100 servers [32]. While for Llama-3 70B, we use a cluster with 16 DGX-H100 servers [31]. In both setups, each server has 8 GPUs with 80GB of high bandwidth memory. The GPUs within a server are connected with NVLINK. Cross-server connection is via InfiniBand.

5.2 Capacity Evaluation

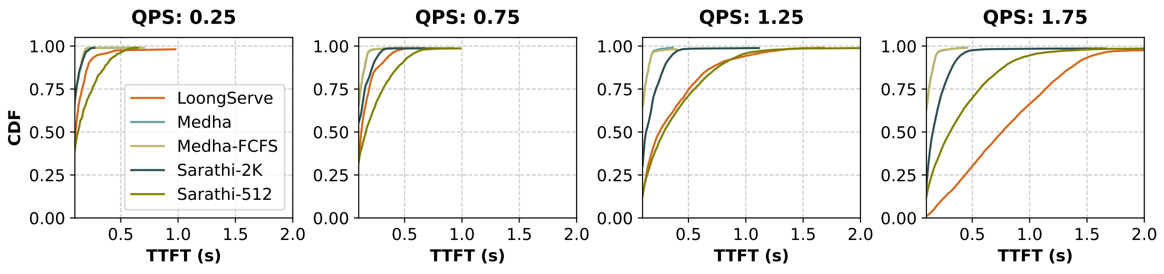
We begin by evaluating how Medha performs under varying loads compared to existing approaches for Llama-3 8B model on the A100 cluster. Our capacity evaluation focuses on two key metrics: TTFT and TPOT, as these directly impact user experience in interactive scenarios.

To evaluate capacity systematically, we designed two workload scenarios: (1) a baseline with only short-context requests (*i.e.*, ShareGPT4) and (2) a mixed workload containing 5% long-context requests (128K–1M tokens). We vary the system load from 0.25 to 1.75 queries per second (QPS) and compare Medha against LoongServe (TP-2, CP-4) and Sarathi (TP-8, PP-2). For fairness, we configure Medha with similar configuration (TP-8, SPP-2).

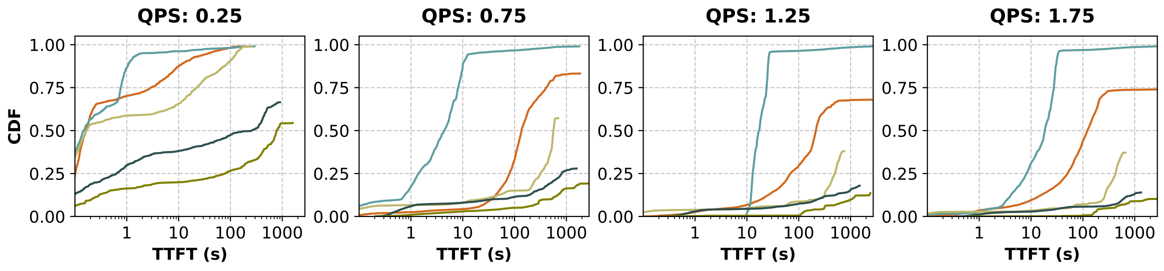
Baseline Performance. In the scenario with only short requests (Figure 15a, Figure 16a), all systems exhibit comparable performance at low loads (0.25 QPS). However, as load increases, LoongServe's performance degrades considerably, which we attribute to resource fragmentation. At 1.75 QPS, LoongServe's P90 TTFT increases dramatically, while Medha maintains consistent latency. Furthermore, Medha achieves considerably better latency compared to Sarathi due to Medha's SPP technique, which helps reduce TTFT.

Long Query Performance. Figure 15b shows significant benefits for Medha with long-context requests. At 0.75 QPS, Medha achieves a 30× median TTFT improvement over LoongServe. Sarathi and Medha-FCFS quickly degrade due to the HOL blocking. Even at 1.25 QPS, Medha maintains acceptable TTFT latencies, offering 5× higher effective capacity than the baselines. Some baseline systems fail to complete requests within the 60-minute profiling window due to HOL blocking, resulting in truncated CDFs.

Decode Performance. Figure 16 shows that LoongServe experiences 5× higher TPOT latencies than Medha, even at high loads without long requests, due to resource fragmentation. With long requests, Medha achieves comparable or better TPOT while processing significantly more requests with an order of magnitude lower TTFT. Even Sarathi, optimized for

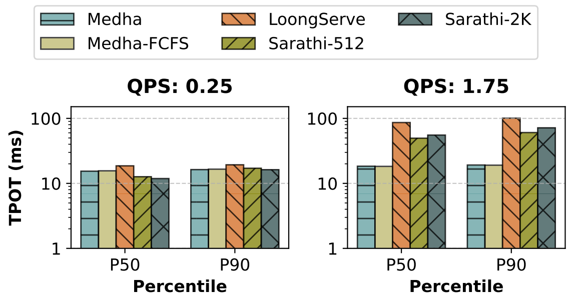


(a) For short-context workloads from ShareGPT4, Medha maintains consistently low latency even at high QPS.

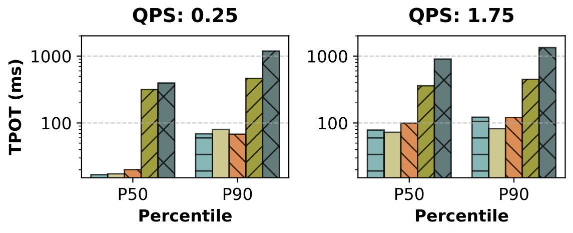


(b) For ShareGPT4 with 5% long requests, Medha achieves up to 30× lower median TTFT, demonstrating effective mitigation of HOL blocking

Figure 15. TTFT latency distribution under varying load conditions for Llama-3 8B on two servers with a total of 16 A100 GPUs.



(a) ShareGPT4.



(b) ShareGPT4 with 5% long requests.

Figure 16. Decode latency for Llama-3 8B on 16 A100s. Due to adaptive chunking, Medha maintains low decode latency while other chunked prefill-based systems suffer from high latency.

low decode latency, reaches TPOTs as high as 1 second due to its static chunking approach, which increases costs for processing later chunks in long sequences. In contrast, Medha's adaptive chunking maintains consistent performance across varying sequence lengths.

5.3 3D Parallel Performance

With Medha's baseline established, we evaluate 3DP using Llama-3 70B on our H100 cluster. We compare two setups with equal resource budgets: (1) a 2D configuration (SPP-8) and (2) a 3D configuration (SPP-4, KVP-2), both using TP8. This isolates the effect of KV cache parallelism. We run a mixed workload, including 5% long-context (2M token) requests, scaled from the Medha-SWE trace.

Figure 17 shows TTFT distributions under varying loads. At lower request rates (0.25 and 0.75 QPS), both configurations perform similarly, with nearly identical CDF curves. At higher loads (1.25 and 1.75 QPS), a trade-off emerges: the 3D parallel setup offers slightly lower peak throughput due to a higher SPP degree, which is more communication-efficient than KVP and better accelerates prefill. Despite this, both configurations maintain similar median latencies.

Figure 18 shows the strength of 3DP in the decode phase. At high load (1.75 QPS), the 3DP setup reduces TPOT by over 2× at both P50 and P90. Even small prefill chunks can delay co-batched decode requests, especially with 2M-token sequences and large models. KVP mitigates this by distributing KV cache reads, reducing decode latency.

This confirms a core design goal of Medha's 3D parallelism: balancing prefill throughput with decode responsiveness. While the 2D setup favors prefill speed, 3DP delivers more consistent end-to-end latency—critical for real-world deployments. It retains the benefits of SPP while combining the strengths of both approaches.

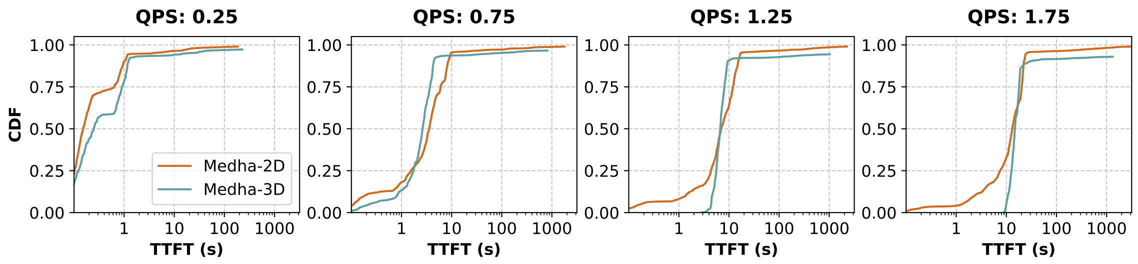


Figure 17. Impact of the parallelization strategy on TTFT distribution across different load points for Llama-3 70B on 8 servers with a total of 64 H100 GPUs running ShareGPT4 with 5% long requests. Both Medha-2D (SPP+TP) and Medha-3D (SPP+TP+KVP) maintain comparable TTFT performance but enable significantly better decode performance by distributing KV cache reads.

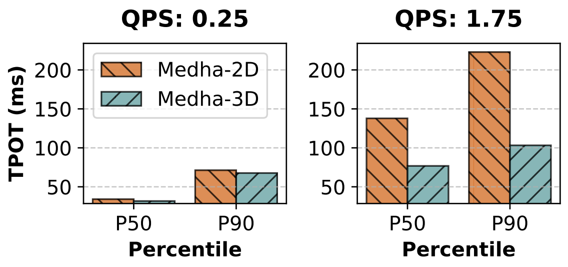


Figure 18. Comparison of decode performance between parallelization strategies for Llama-3 70B with 5% long requests. Medha-3D's KV cache parallelism delivers 2× compared to Medha-2D.

5.4 Effectiveness of Medha Scheduler

End-to-End. Figure 21 shows the number of active requests in the system for both LoongServe and Medha. In LoongServe, HOL blocking and pool fragmentation lead to a sawtooth pattern, as small prefill requests are frequently delayed behind longer ones. In contrast, Medha maintains a stable number of active requests through fine-grained preemption and SLO-aware scheduling.

Scheduler Ablation. We isolate the impact of Medha's optimized slack-based prefill prioritization and prefill-prefill batching by comparing it to traditional scheduling policies. Figure 22 shows the TTFT distributions for four approaches: FCFS, EDF, LRS (with slack-based prioritization), and Medha's scheduler with prefill-prefill batching. The evaluation uses Llama-3 8B on A100 GPUs in TP8-SPP2 configuration with a mixed workload of 5% long-context requests.

At low load (0.25 QPS), all policies show similar median latency but differ in tail behavior. However, at high load (1.75 QPS), the differences become more pronounced. FCFS performs poorly due to unmitigated HOL blocking from long requests. Despite its success in latency-sensitive systems, EDF struggles here. While effective at low loads, EDF's performance degrades at higher loads, resembling FCFS behavior. This occurs because EDF defers long requests until their deadlines become unfeasible, causing them to be prioritized only once they pass their deadlines.

We also compare Medha to our adaptation of LRS with normalization to handle request length heterogeneity. The key difference between Medha and LRS is space sharing. While LRS reduces HOL blocking compared to FCFS and EDF, it results in up to 1.8× higher median latency than Medha due to the lack of space sharing.

Pool Fragmentation. A common industry technique to mitigate HOL blocking when serving models with moderate context lengths (64-128K) is to create separate pools for short and long requests. While LoongServe dynamically creates similar pools based on prefill lengths, it does not guarantee the availability of dedicated resources for all short requests. To evaluate the effectiveness of this approach, we implement a version of LoongServe with a *reserved pool* specifically for short request processing, as shown in Figure 23.

We compare Medha to this baseline using the same setup as Section 5.2, reserving two of eight CP instances for short requests (<8192 tokens) and the rest for long requests. Each pool uses the standard LoongServe scheduler. This reservation increases contention for long prefills, leading to up to 20% lower completions for long requests compared to Medha, and 10% lower than default LoongServe. For the decodes, LoongServe with reservation achieves slightly lower TPOT compared to LoongServe as an artifact of overall lower ingestion (prefill) rate. Medha consistently achieve lower decode latency compared to be both the variants of LoongServe.

5.5 Scaling Efficiency

The ultimate measure of Medha's effectiveness is its ability to maintain high throughput while scaling to large parallelism degrees. We evaluate this using hardware utilization metrics Model FLOPS Utilization (MFU) and Model Bandwidth Utilization (MBU) [1, 10]. In LLM inference, prefill phases are compute-bound while decode phases are memory-bound [35, 36]. Figure 19 shows the MFU for Medha in the prefill phase (2D SPP+TP), while Figure 20 shows the MBU for the decode phase (2D KVP+TP). For Llama-3 70B, we achieve 50–60% MFU across configurations, improving for longer sequences. Even at the scale of 128 GPUs, we achieve over 50% MFU. Examining MBU, Figure 20 shows

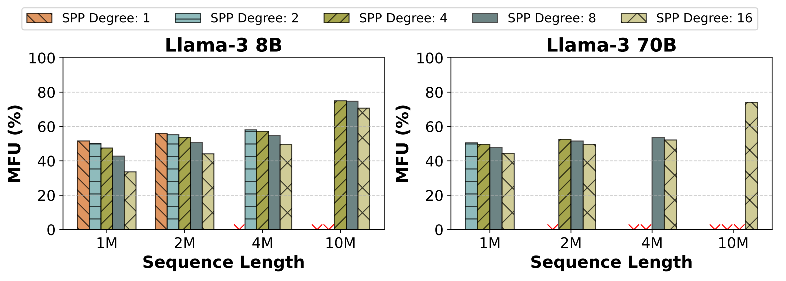


Figure 19. Model FLOPS Utilization [10] (MFU) for Medha 2D (TP+SPP). It achieves 50-60% utilization across sequence lengths and parallelism degrees.

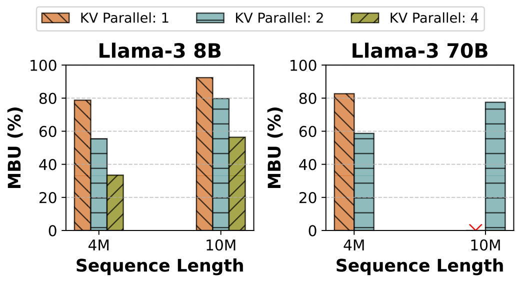


Figure 20. Model Bandwidth Utilization (MBU) for Medha 2D (TP+KVP).

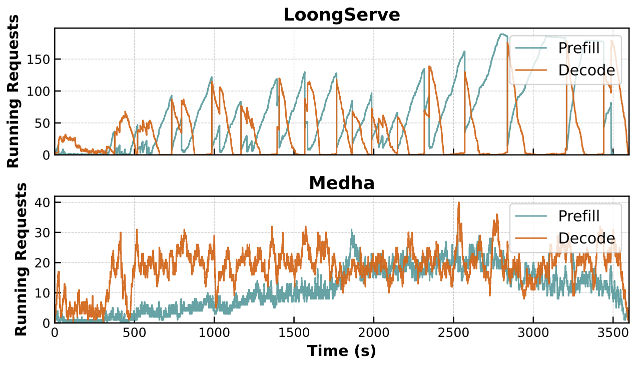


Figure 21. Active requests over time in LoongServe and Medha using the 5% long request trace. Note that active requests also include pending/queued requests from HOL blocking.

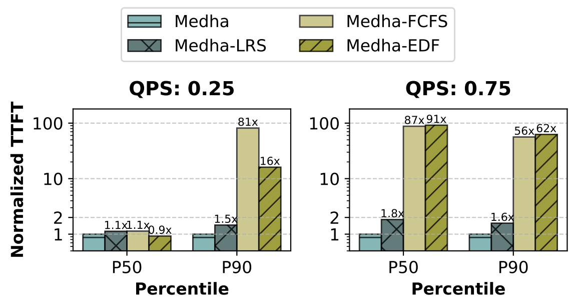
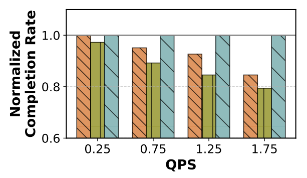


Figure 22. Impact of different scheduling policies on normalized TTFT latency. Even compared to our modified LRS policy, Medha scheduler achieves (1.6–1.8×) lower latency, demonstrating the effectiveness of Medha's prefill-prefill batching technique.

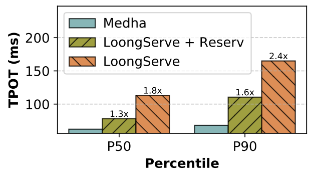
that Medha's KVP implementation achieves up to 92% MBU in optimal configurations, allowing consistent decode performance even with extremely long contexts.

6 Related Work

LLMs for long context. Recent research has focused on effectively training and serving long-context LLM models. Some propose new attention parallelism techniques as more efficient solutions to enable long context [9, 25, 28]. We discuss and compare them in detail in Sections 2 and 5. A similar idea to SPP without adaptive chunking, called token-parallelism, was used in TeraPipe [26] to parallelize the different micro-batches of a mini-batch along the token dimension in order to reduce pipeline bubbles and improve throughput during training. Medha creates small mixed-batches of chunked prefill and decodes and then parallelize these mixed batches to maintain latency targets during inference.



(a) Normalized completion rate.



(b) Decode latency at 0.75 QPS.

Figure 23. Impact of pool fragmentation on long requests for Llama-3 8B on 16-A100 with 5% long requests. Medha maintains maximum throughput and lowest latency. Adding a dedicated reserved pool to LoongServe (+Reserv) to mitigate HOL blocking for short requests fragments resources and further degrades overall completion rate for long requests compared to both standard LoongServe.

Approximate alternatives. State Space Models (SSMs) [15, 16] offer alternative attention-based architectures to reduce computational complexity. Other techniques like locality-sensitive hashing (LHS) [22], compressive attention [34], and prompt/KV cache compression [21, 24, 56] reduce computation and memory footprint. While these methods trade accuracy for efficiency, we focus on transformer models that *preserve* accuracy by retaining the full context. Medha can also be combined with approximate techniques.

Request scheduling. Efficient request scheduling has been extensively studied [13, 29, 38, 39, 42, 45, 50], but existing approaches have notable limitations when addressing long-context requests. For example, SRTF scheduling [13, 39] reduces median latency but leads to starvation of long requests due to lack of preemption. LoongServe[49] supports space sharing among concurrent long requests but lacks preemption and time-sharing, resulting in significant HOL delays, especially under FCFS scheduling. Fairness-focused schedulers like [42] emphasize equitable resource distribution among clients but fail to address strict latency SLOs. In contrast, Medha introduces a slack-based fine time sharing scheduling policy with prefill-prefill batching, enabling efficient mixing of long and short requests to meet latency SLOs.

7 Conclusion

We present Medha, an efficient and scalable long-context LLM inference system that combines novel adaptive chunking and 3D Parallelism techniques to achieve fast prefill and decode up to 10M tokens. By incorporating variable context support, mixed batching, and a slack-aware scheduling policy with space-time sharing, Medha dynamically prioritizes requests based on performance SLOs. This design improves throughput and resource efficiency while ensuring latency guarantees across diverse workloads, making Medha a practical solution for long-context interactive LLM applications.

References

[1] [n. d.]. LLM Inference Performance Engineering: Best Practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices.

[2] [n. d.]. ZeroMQ. https://zeromq.org/.

[3] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. *MLSys* (2024).

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. *OSDI* (2024).

[5] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG]

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL]

[7] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2023. L-eval: Instituting standardized evaluation for long context language models. *arXiv preprint arXiv:2307.11088* (2023).

[8] Sparsh Bhasin. 2024. Enhancing LLM Context Length with RoPE Scaling. https://blog.monsterapi.ai/blogs/enhancing-llm-context-length-with-rope-scaling.

[9] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. 2023. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431* (2023).

[10] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[11] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691 [cs.LG]

[12] Apache Foundation. [n. d.]. Apache Ray. https://docs.ray.io/en/latest/index.html.

[13] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient LLM Scheduling by Learning to Rank. *arXiv preprint arXiv:2408.15792* (2024).

[14] Google. 2024. Gemini – Long context. https://ai.google.dev/gemini-api/docs/long-context

[15] Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* (2023).

[16] Albert Gu, Karan Goel, and Christopher Ré. 2021. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396* (2021).

[17] 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. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. arXiv:2401.08671 [cs.PF]

[18] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2023. FlashDecoding++: Faster Large Language Model Inference on GPUs. *arXiv preprint arXiv:2311.01282* (2023).

[19] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv preprint arXiv:2401.11181* (2024).

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

[21] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LongLLMLingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839* (2023).

[22] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451* (2020).

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*.

[24] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *OSDI*.

[25] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2021. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120* (2021).

[26] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. *arXiv preprint arXiv:2102.07988* (2021).

[27] Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. 2024. World Model on Million-Length Video And Language With Blockwise RingAttention.

[28] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. arXiv:2310.01889 [cs.CL] https://arxiv.org/abs/2310.01889

[29] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. 2024. Andes: Defining and Enhancing Quality-of-Experience in LLM-Based Text Streaming Services. *arXiv preprint arXiv:2404.16283* (2024).

[30] Meta. [n. d.]. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence.

[31] Microsoft Azure. 2024. ND-H100-v5 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndh100v5-series?tabs=sizenetwork.

[32] Microsoft Azure. 2024. NDm-A100-v4 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-

accelerated/ndma100v4-series?tabs=sizebasic.

[33] Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867* (2018).

[34] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. 2024. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143* (2024).

[35] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. In *ISCA*.

[36] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. 2023. POLCA: Power Oversubscription in LLM Cloud Providers. arXiv:2308.12908 [cs.DC] https://arxiv.org/abs/2308.12908

[37] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, and Xinran Xu Weimin Zheng. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:2407.00079 [cs.DC] https://arxiv.org/abs/2407.00079

[38] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Power-aware Deep Learning Model Serving with μ-Serve. In *USENIX Annual Technical Conference (USENIX ATC)*.

[39] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*.

[40] M Reid, N Savinov, D Teplyashin, Lepikhin Dmitry, T Lillicrap, JB Alayrac, R Soricut, A Lazaridou, O Firat, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[41] Rya Sanovar, Srikant Bharadwaj, Renee St Amant, Victor Rühle, and Saravan Rajmohan. 2024. Lean Attention: Hardware-Aware Scalable Attention Mechanism for the Decode-Phase of Transformers. *arXiv preprint arXiv:2405.10480* (2024).

[42] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. 2023. Fairness in Serving Large Language Models. *arXiv preprint arXiv:2401.00588* (2023).

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

[44] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024).

[45] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *OSDI*.

[46] Gradient team. [n. d.]. Scaling Rotational Embeddings for Long-Context Language Models. https://gradient.ai/blog/scaling-rotational-embeddings-for-long-context-language-models.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[48] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. 2023. OpenChat: Advancing Open-source Language Models with Mixed-Quality Data. arXiv:2309.11235 [cs.CL]

[49] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-context Large Language Models with Elastic Sequence Parallelism. In *SOSP*.

[50] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for Large Language models. *arXiv preprint arXiv:2305.05920* (2023).

[51] An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, Weijia Xu, Wenbiao Yin, Wenyuan Yu, Xiafei Qiu, Xingzhang Ren, Xinlong Yang, Yong Li, Zhiying Xu, and Zipeng Zhang. 2025. Qwen2.5-1M Technical Report. *arXiv preprint arXiv:2501.15383* (2025).

[52] Amy (Jie) Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, GrigorySizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. 2024. Context Parallelism for Scalable Million-Token Inference. *arXiv preprint arXiv:2411.01783* (2024).

[53] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. 2024. Accelerating Self-Attentions for LLM Serving with FlashInfer. https://flashinfer.ai/2024/02/02/introduce-flashinfer.html

[54] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *OSDI*.

[55] Tao Yuan, Xuefei Ning, Dong Zhou, Zhijie Yang, Shiyao Li, Minghui Zhuang, Zheyue Tan, Zhuyu Yao, Dahua Lin, Boxun Li, et al. 2024. Lv-eval: A balanced long-context benchmark with 5 length levels up to 256k. *arXiv preprint arXiv:2402.05136* (2024).

[56] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Re, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Conference on Parsimony and Learning (Recent Spotlight Track)*. https://openreview.net/forum?id=w4IRMAJYPk

[57] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. arXiv:2401.09670 [cs.DC]