No Request Left Behind: Tackling Heterogeneity in Long-Context LLM Inference with Medha

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

Abstract

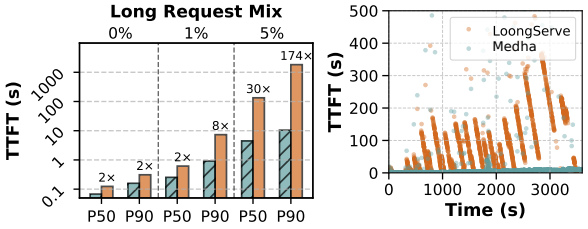
Deploying million-token Large Language Models (LLMs) is challenging because production workloads are highly heterogeneous, mixing short queries and long documents. This heterogeneity, combined with the quadratic complexity of attention, creates severe convoy effects where long-running requests stall short, interactive ones, degrading system responsiveness. We present MEDHA, a serving system that eliminates these convoys by introducing fine-grained, preemptive scheduling to LLM inference.

MEDHA makes preemption practical with a co-designed set of mechanisms – including *Adaptive Chunking* and *Stream Pipeline Parallel*– that overcome the perceived inefficiencies and scaling challenges of chunking. Additionally, we present a new parallelism strategy *KV-Cache Parallelism* to reduce the decode latency and afford interactivity despite very long context. These mechanisms are orchestrated by a *Length-Aware Relative Slack (LARS)* scheduler, a deadline- and heterogeneity-aware scheduling policy that prevents both the convoy effect and the starvation that plagues simpler policies. Under a heterogeneous workload, MEDHA improves throughput by 5.7× while reducing median and 99th-percentile latency by 30× and 174×, respectively, compared to state-of-the-art non-preemptive systems.

1 Introduction

Large language models with million-token context windows are transforming how we interact with information – enabling large-scale document analysis, multi-hour video understanding [16, 24, 45], and autonomous coding agents [9, 17]. However, deploying these models in production poses a significant challenge: real-world workloads combine both long and short requests. A single service must handle everything from 100-token chat messages to 10M-token document processing, often from the same users within the same session.

Motivation. This mix of request lengths to the same model instance creates extreme computational heterogeneity due to the quadratic complexity of self-attention [40]. A 100K-token request is not 100× but approximately 10,000× more computationally expensive than a 1K-token request. When these requests share the same serving infrastructure, we run into severe performance degradation due to: the *convoy effect* [10, 37], where long-running requests block shorter ones, resulting in poor system responsiveness. As shown in Figure 1, even with just 5% long requests in the workload, state-of-the-art systems like LoongServe [42] experience 30× median latency increases and 174× tail latency degradation for short requests.



(a) TTFT distribution where state-of-the-art baseline shows 30-174× higher latency due to lack of preemptive scheduling.

(b) TTFT over time showing severe convoy effect 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 [42] and MEDHA at 0.75 QPS.

Context Parallelism (CP) [11, 22] has made it possible to distribute long-context processing across hundreds of GPUs, enabling effective training for long-context models. LoongServe [42] and Yang et al. [44] adapted these techniques for inference by introducing elasticity to context parallelism, dynamically adjusting the degree of parallelism based on request length to improve resource efficiency. However, these approaches fundamentally lack preemptability — once a long request begins processing, it cannot be interrupted until completion. This non-preemptive execution model inevitably results in convoy effects in heterogeneous workloads. Fig-

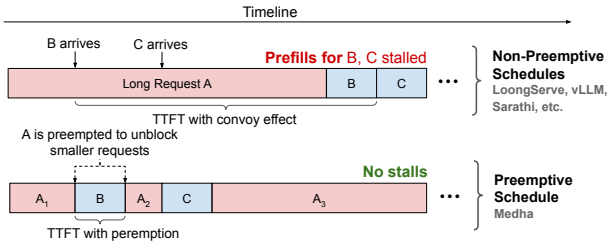


Figure 2: Impact of preemption on convoy effect. Non-preemptive scheduling (top) blocks short requests B and C behind long request A, causing deadline violations. Preemptive scheduling (bottom) interleaves execution through chunking, eliminating convoy effect while maintaining throughput.

ure 1b demonstrates this problem empirically: LoongServe exhibits severe latency spikes lasting for 100s of seconds whenever long requests block the system. These spikes occur because arriving short requests must wait for the entire prefill computation of the long request to complete — which can take several minutes.

The convoy effect is a well-studied problem in operating systems with a known solution: preemptive scheduling. Chunked prefills [6] provide a natural mechanism for preemptable prefill computation. However, applying preemption to LLM inference faces three barriers that have discouraged its adoption. First, chunking long prefills are considered inefficient due to ***repeated KV-cache reads.*** Second, batching decodes requests with chunked prefills for long requests results in ***high decode latency*** that degrades user experience. Finally, existing context parallelism techniques for long-context inference are ***fundamentally incompatible with chunked execution***.

Our work. MEDHA makes preemptive long-context inference practical by systematically addressing each barrier. We demonstrate that KV-cache read amplification is a non-issue for modern architectures – chunks as small as 40 tokens achieve near-optimal efficiency due to high arithmetic intensity in grouped-query attention. We introduce *adaptive chunking*, which dynamically adjusts chunk sizes as the computational bottleneck shifts from MLP to attention operations, maintaining both high throughput and predictable latency. We develop two parallelism strategies compatible with preemption: ***Stream Pipeline Parallelism (SPP)*** accelerates prefills by pipelining chunks across stages, while ***KV-Cache Parallelism (KVP)*** bounds decode latency by distributing attention computation.

To effectively leverage the preemptable prefills, we introduce Length-Aware Relative Slack (LARS), a deadline-aware scheduling policy designed to explicitly tackle the heterogeneous nature of long-context inference. Unlike traditional policies that either cause convoy effects (First Come First Serve - FCFS) or starvation (Earliest Deadline First - EDF, Least Remaining Slack - LRS), LARS ensures both short and long requests meet their deadlines by pushing completions toward their SLO boundaries – maximizing schedule robustness against unpredictable arrivals. Furthermore, we introduce a dynamic batch packing algorithm that creates batches that maximally utilize GPU compute by co-locating complementary prefill chunks and decode requests while respecting strict time budgets.

MEDHA unifies these techniques in a unified serving system that scales to multi-million token requests while maintaining high throughput and low latency. In summary, we make the following contributions in this paper:

- We identify and quantify the convoy effect in long-context inference arising due to request-length heterogeneity in current non-preemptive systems.
- We make chunked prefills viable for preemptive long-context inference by systematically addressing their perceived inefficiencies: introducing adaptive chunking to balance throughput and latency, and developing Stream Pipeline Parallelism (SPP) and KV-Cache Parallelism (KVP) as preemption-compatible parallelism strategies.
- We propose Length-Aware Relative Slack (LARS), a scheduling policy that prevents convoy effects and starvation by accounting for workload heterogeneity and user SLOs.
- We implement MEDHA with optimized kernels and scheduler design, demonstrating 5.7× higher throughput and up to 174× lower tail latency than state-of-the-art non-preemptive systems on real long-context workloads.

2 Motivation: The Case for Preemptive Long Context LLM Inference

In this section, we analyze how million-token LLM inferences create extreme computational heterogeneity due to quadratic attention complexity, causing convoy effects where long requests block shorter ones, motivating our preemptive inference approach.

2.1 Background: LLM Inference Characteristics

Auto-regressive LLM inference comprises two fundamentally different phases with distinct performance characteristics [7, 29, 49]. The **prefill phase** processes the entire prompt through a single forward pass to construct Key-Value (KV) cache and generate the first token. This phase is compute-bound, with performance measured by Time-to-First-Token (TTFT). The subsequent **decode phase** generates tokens autoregressively, one at a time, and is memory-bandwidth-bound. Its performance is measured by Time-Per-Output-Token (TPOT) or Time-Between-Tokens (TBT) [4].

Contemporary serving systems employ two primary parallelism strategies. **Tensor Parallelism (TP)** [36] partitions each model layer across multiple devices. This reduces per-

device memory requirements and can improve latency. However, TP requires high communication bandwidth, limiting it to single servers with fast interconnects like NVLink. **Pipeline Parallelism (PP)** [7, 18, 47] distributes complete layers sequentially across devices. While this reduces memory pressure per device and can improve throughput, it provides no latency benefit for individual requests due to its sequential execution model.

2.2 Context Length Scaling Limits of Conventional Parallelism Techniques

Memory Constraint. In the prefill phase, since all the input tokens are processed concurrently, the activation memory required for prefill computation increases linearly with the context length. While tensor parallelism can distribute this load across devices, it cannot be scaled beyond a single node due to communication overhead.

Latency Constraints. The quadratic complexity of attention operation becomes a major challenge for interactive workloads as sequence length grows. For instance, to process 1M tokens with Llama-3 70B we require a total of 2.8 ExaFLOPs. On an H100 GPU, even at full utilization, this computation would require at least 48 minutes to execute. To perform this computation in a reasonable time, the attention computation needs to be parallelized across a large number of GPUs. However, neither tensor nor pipeline parallelism provides a viable solution. As discussed previously, TP does not scale beyond a single node (8 GPUs) due to communication overhead [7, 28], while PP can scale to a large number of GPUs, it does not provide any latency advantage.

> ***Takeaway:*** Standard parallelization techniques like TP or PP fail at million-token contexts due to memory limits and quadratic attention costs that lead to high latency.

2.3 Long-Context System Scaling with Context Parallelism

To overcome the memory and latency limitations of conventional parallelism, Liu et al. introduced **Context Parallelism (CP)** [11, 22] for long-context *training*. In CP, the input sequence is partitioned across multiple GPUs to alleviate activation memory pressure. By overlapping KV block communication between GPUs with computation, CP enables efficient scaling to hundreds of devices. This approach has been widely adopted in long-context training systems.

However, context parallelism's design is fundamentally misaligned with the demands of inference serving. To achieve efficient overlap of communication and computation in CP, each GPU must process a sufficiently large sequence partition (e.g., 24.5K tokens on A100 with InfiniBand [22]). This creates a critical latency-throughput tradeoff — a system configured with high parallelism degrees for low-latency serving of long context requests suffers from severe underutilization when serving short requests. Conversely, a system configured for short requests cannot achieve acceptable latency for long ones.

2.4 Adapting Context Parallelism for Inference

To address the rigid resource allocation in CP, the state-of-the-art system LoongServe [42] adapts it for inference by introducing two key mechanisms. First, it proposes an elastic version of context parallelism, where the degree of parallelism is dynamically adjusted to match the workload — allocating more resources to accelerate long, compute-intensive prefills while using fewer for short requests, thereby improving efficiency.

Second, because CP is ineffective for decode phases, the system must adopt the prefill-decode disaggregation paradigm [29, 49]. In this model, the prefill and decode phases are handled by separate, isolated groups of GPUs. After a request's prefill is complete, its KV cache is migrated from the prefill pool to a different group of GPUs dedicated to the less resource-intensive decode phase. Furthermore, since the relative prefill to decode load ratio in the system dynamically changes based on the input requests pattern [27], LoongServe adopts an elastic approach where the number of GPUs in the prefill/decode pool is dynamically adjusted to match the workload. This elastic, disaggregated architecture represents the current state-of-the-art approach for long-context inference — achieving 3-5× [42] lower latency than prior systems like vLLM [19], DistServe [49], and Sarathi-Serve [6].

2.5 The Convoy Effect from Extreme Workload Heterogeneity

The key challenge in serving long-context models stems from the extreme workload heterogeneity created by the quadratic complexity of self-attention. Because the required FLOPs for the attention mechanism scale with the square of the sequence length (N^2), the difference in processing time between requests grows superlinearly. For instance, a 100K-token request is not 100× but roughly 10,000× more computationally expensive than a 1K-token request. This extreme heterogeneity, lead to a classic systems challenge known as the ***convoy effect*** [10, 37]. When the system processes a long prefill, all subsequent short requests behind it in the queue are stalled. As shown in Figure 1a, this leads to a complete collapse in system performance, increasing median TTFT by 30× and tail latency by 174× with just 5% long requests in the workload.

Takeaway: The quadratic cost of attention creates extreme workload heterogeneity, leading to the ***convoy effect***, where long requests block short ones.

2.6 The Path to Preemption: Fine-Grained Chunking

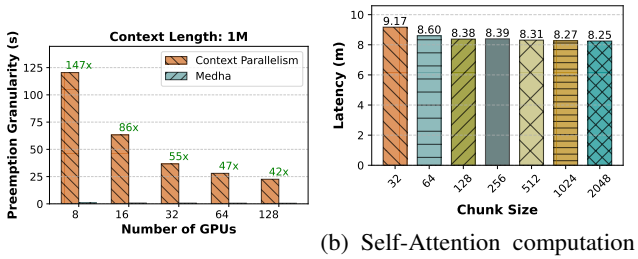
The convoy effect is a widely studied problem in operating systems — resolving this issue requires a shift from non-preemptive to preemptive scheduling [10, 37]. To apply this principle to LLM serving, the long, atomic prefill operation must be broken down into smaller, interruptible units of work. Chunking the input prompt [6] achieves exactly this, creating scheduling opportunities to interleave short requests with long ones.

However, naive chunking is widely considered impractical for long contexts due to three prohibitive systems challenges, which this paper systematically resolves:

KV-Cache Read Amplification. In a standard, non-chunked prefill, the KV cache is read once. With chunking, however, the processing of each subsequent chunk requires re-reading the entire KV cache generated by all previous chunks from GPU memory. This transforms the memory access pattern from being linear with the sequence length to being quadratic. Because memory bandwidth is a critical and often limited resource, this quadratic increase in data movement has led to the widespread belief that chunking is fundamentally inefficient and unscalable for long-context serving [7, 49].

Latency Interference. Piggybacking prefill chunks [6] onto decode batches is a standard technique to improve GPU utilization and reduce tail latency by co-executing compute-bound prefill operations with memory-bound decode operations. However, with long contexts, the compute cost of successive prefill chunks grows quadratically as the context lengthens. Late-stage chunks become so computationally intensive that they stall latency-sensitive decodes, making it infeasible to batch prefill chunks with latency-sensitive decodes — for instance, computing the a prefill chunk for a 1M context request with Llama-3 8B on 8 H100s using chunk size of 512 results in decode latency of $\sim$ 250ms, almost an order of magnitude higher the typical production SLOs.

Lack of a Preemption-Friendly Scaling Strategy. Adopting chunking means abandoning the only proven technique for scaling prefill latency – context parallelism. While CP operates by splitting the sequence in a special dimension across different devices, chunked prefill unrolls the prefill computation in a temporal dimension by processing each prefill chunk sequentially. There is no trivial solution to combine the two approaches. The conventional alternatives are insufficient for serving long contexts. This creates the need for entirely new parallelism strategies designed to be both scalable and preemptive.



(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 3: Efficacy of chunked prefill for long-context inference.

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
I	arithmetic intensity
c	chunk size

3 Enabling Efficient Preemptable Prefills

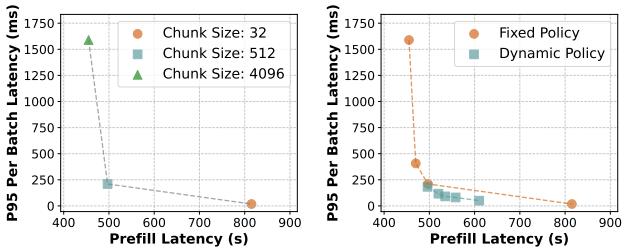
To enable preemptive execution using chunked prefills we need to overcome three perceived barriers: the belief that chunking causes prohibitive KV-cache read amplification, concerns about latency interference between chunked operations, and the lack of preemption-friendly parallelism strategies. In this section, we present the insights that allow MEDHA to systematically address these challenges.

3.1 Debunking KV-Cache Read Amplification Inefficiency Myth

Conventional wisdom maintains that chunked attention is inherently inefficient due to KV-cache read amplification—the repeated reading of cached keys and values across chunks. We analyze the attention computation from first principles and demonstrate this assumption is incorrect for modern model architectures.

Arithmetic Intensity Analysis. Modern GPU architectures feature independent compute and memory subsystems that operate concurrently in a pipelined fashion. Performance is determined by whichever subsystem becomes saturated first. When the compute subsystem is fully utilized, additional memory operations execute in parallel *"for free"* without impacting the device throughput.

The key metric determining this behavior is arithmetic intensity—the ratio of compute operations to memory accesses. High arithmetic intensity indicates sufficient computation per byte of data to keep compute units busy while



(a) Static chunk sizes. (b) Adaptive chunk size.

Figure 4: 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.

memory transfers complete in parallel. For chunked attention, this relationship is governed by:

$$I^i_{cp}(n,c) \simeq \frac{4ic^2dh_q}{4icdh_{kv}} = c\frac{h_q}{h_{kv}} \tag{1}$$

The critical insight is that arithmetic intensity for chunked attention depends solely on chunk size, not total context length. Each chunk processes c tokens, requiring reads of the full KV cache but performing a fixed number of operations per token. Furthermore, contemporary LLMs employ Grouped-Query Attention architectures where multiple query heads share KV heads (e.g., 8× in Llama-3 70B), resulting in high arithmetic intensity such that even small chunks can saturate GPU compute.

Empirical Validation. On H100 GPUs running Llama-3 70B, chunks of just 40 tokens can saturate GPU compute. We find that for 1M-token contexts, 32-token chunks incur merely 11% overhead relative to 2048-token chunks as shown in Figure 3b. Note that, unlike older attention implementations used prior analysis [7] — where the chunked prefill was shown to be inefficient for long contexts — modern attention kernels like FlashInfer and FlashAttention-2 [13, 46] parallelize over both query and KV dimensions, which helps in materializing the theoretical performance potential of the chunked prefill.

3.2 Managing Interference with Adaptive Chunking

Having established that chunking is computationally efficient, we now address the prefill-decode latency interference in mixed batches.

The Throughput-Latency Tradeoff of Chunking. Chunked prefills face a fundamental throughput-latency tradeoff. To maximize system throughput, a scheduler must use large chunks to process prefills efficiently; however, to guarantee low decode latency for co-batched requests, it must use small chunks. This tradeoff would be trivially resolved if we could execute small chunks with minimal overhead. As shown in section 3.1, even chunks as small as 40 tokens are enough to saturate *attention* computation — however, the challenge arises because different operations show varying performance characteristics. The MLP component has a significantly lower arithmetic intensity than attention — c as opposed to $c\frac{h_q}{h_{kv}}$ for attention, and is thus more sensitive to chunk size. Moreover, there are fixed per-chunk overheads like kernel launches that are not amortized by chunking. As shown in Figure 4a, this conflict forces an undesirable choice between high throughput and low decode latency.

Resolving the Tradeoff with Adaptive Chunking. To resolve the throughput-latency tradeoff, our approach is based on the key insight that a long prefill's computational bottleneck is not static, but *shifts* as the prefill progresses. Prefill computation is initially dominated by MLP layers, which require large chunks to run efficiently and achieve high throughput. As the KV cache grows, the quadratic cost of the attention operation becomes the overwhelming dominant cost. At this stage, a switch to smaller chunks is possible. While smaller chunks make the MLP computation slightly less efficient, this is an acceptable trade-off because the performance hit is negligible compared to the now-dominant cost of attention. Based on this insight, MEDHA implements an **Adaptive Chunking** policy. The policy begins a prefill with large chunks and dynamically shrinks them as the bottleneck shifts, thereby maintaining a predictably low iteration time. This adaptive strategy resolves the tradeoff faced with static chunking, achieving both high prefill throughput and low decode latency, as shown in Figure 4.

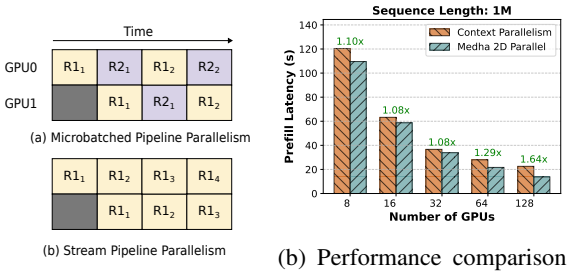
3.3 Scalable Parallelism for Preemptive Inference

To achieve interactive latency for million-token requests, preemptive chunking must be combined with a scalable, multi-node parallelism strategy. As existing approaches are incompatible with our preemptive model, MEDHA introduces two novel techniques: Stream Pipeline Parallelism and KV-Cache Parallelism.

Accelerating Prefill Computation. We leverage an overlooked opportunity to accelerate prefill computation: while prior works process each chunk sequentially through all model layers, we observe that the chunks of a single request can be processed concurrently across pipeline stages.

In chunked prefills, chunk $i+1$ requires the KV cache from chunk i, but critically, it does *not* need chunk i's final model output. This means chunk $i+1$ can begin processing as soon as chunk i completes the first pipeline stage; it does not need to wait for chunk i to finish all pipeline stages.

Traditional approaches that combine chunking with pipeline parallelism treat each chunk like a separate request,



(a) Contrasting PP strategies for prefill processing.

(b) Performance comparison of Context Parallelism vs SPP+TP.

Figure 5: Microbatched pipeline parallelism interleaves micro-batches composed of prefills from different requests ($R1$, $R2$) to improve throughput. SPP on the other hand, overlaps chunks of the same request ($R1_1$, $R2_2$) across stages to accelerate prefill processing. SPP achieves better scaling compared to CP due to lower communication overhead, resulting in up to 1.64× lower prefill latency for 1M context processing for Llama-3 8B with H100s.

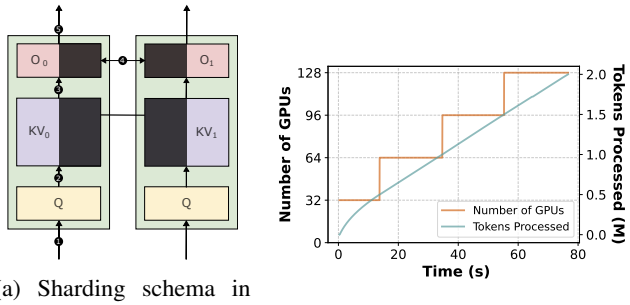
processing them sequentially through the entire pipeline. This leaves pipeline stages underutilized. For example, when chunk i moves from Stage 1 to Stage 2, Stage 1 sits idle. To fill these "pipeline bubbles," different requests are interleaved across stages, a technique known as micro-batching.

In contrast, MEDHA introduces **Stream Pipeline Parallelism (SPP)**, which exploits the unique data dependency structure of chunked prefills. It schedules chunk $i+1$ to start Stage 1 immediately when chunk i advances to Stage 2 Figure 5a. This allows layers across all pipeline stages to operate concurrently on different chunks of the same long prefill, reducing the critical path of the prefill by the pipeline depth.

This chunk-level pipelining scales effectively. As shown in Figure 5b, Stream Pipeline Parallelism scales nearly linearly with the number of stages, enabling inference on multi-million-token requests using hundreds of GPUs. Please refer to Section B for scaling results up to 10M tokens. Compared to context parallelism, stream pipeline parallelism is significantly faster, achieving 1.64× lower latency on a one-million-token prefill and reducing TTFT over 128 H100s.

Bounding Decode Latency. While SPP addresses prefill latency, the decode phase presents its own scaling challenge. For long contexts, decode latency grows linearly with the sequence length, leading to high TPOT and a poor user experience. Furthermore, in mixed batches, even the smallest efficient prefill chunk (as determined by Adaptive Chunking) can still significantly slow down the decode operation when operating with very long contexts or larger models, creating a need for an additional mechanism to control iteration time.

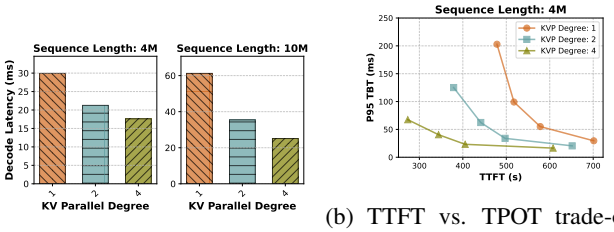
MEDHA introduces **KV-Cache Parallelism (KVP)** as a unified mechanism to address both challenges by parallelizing the KV cache reads across multiple devices along the sequence dimension. During any computation step (either a decode token or a prefill chunk), the query is replicated



(a) Sharding schema in KVP.

(b) KVP Execution Timeline.

Figure 6: KV Parallelism distributes the KV-cache state across GPUs to minimize the latency of long requests. During prefill, KVP dynamically scales resources as the KV produced grows to maintain consistent iteration times irrespective of context length. While processing 2M tokens with Llama-3 8B using $p_{tp} = 8$, $p_{kvp} = 4$, and $p_{spp} = 4$. MEDHA starts with a single KVP worker group (4 servers) and progressively scales up to 16 servers.



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

(b) TTFT vs. TPOT trade-off space.

Figure 7: (a)TPOT reduction with KVP in MEDHA 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. (b) KVP enables co-batching of larger prefill chunks with decode requests by parallelizing attention computation, reducing prefill-decode latency interference and providing a richer tradeoff space.

to each device, which computes a partial attention output in parallel using its local shard. These partial outputs are then combined using online-softmax.

KVP provides two critical, complementary benefits. First, for the decode phase, it places an upper bound on TPOT by ensuring the decode time can be capped for long requests, as shown in Figure 7a. Second, for the prefill phase, KVP offers a new lever to manage latency interference that is complementary to Adaptive Chunking. It allows the scheduler to use larger, more throughput-efficient chunks while still meeting decode SLOs by parallelizing the chunk's internal attention computation, improving the overall TTFT-TPOT tradeoff as shown in Figure 7b. MEDHA uses a *progressive scaling* strategy, dynamically adding KVP workers as the context grows to maintain a near-consistent iteration latency. We provide additional KVP results in the appendix Section B.

Takeaway: When paired with mechanisms for scaling computation, chunked prefills provide a viable foundation for preemptive long-context inference.

4 Scheduling Policies for Preemptive Inference

The mechanisms presented in Section 3 provide the necessary tools for preemptive inference for long-context requests. However, these are not sufficient on their own. To effectively navigate the throughput-latency tradeoff and resolve convoy effects with a mix of long and short requests, a robust scheduling policy is required. Figure 8 illustrates how MEDHA orchestrates these components: the Replica Controller implements our scheduling policies through a slack-aware Batch Scheduler that maintains request priorities using LARS, and a Batch Packer that constructs optimal batches guided by runtime predictions. These batches are then dispatched to the 3D Parallel Execution Engine, which leverages our novel combination of KVP, SPP, and TP to execute them efficiently. This section details the scheduling policies that drive these components—how MEDHA prioritizes requests to prevent convoy effects (Section 4.1), co-locates complementary prefills to improve throughput (Section 4.2), and packs batches to meet strict SLO requirements (Section 4.3).

4.1 An SLO-Aware Prioritization Policy

In this section, we develop an online scheduling policy that prevents convoy effects while avoiding starvation, operating with sub-millisecond overhead. We analyze why widely used policies for LLM inference — FCFS, EDF, and LRS fail under extreme heterogeneity, then present the approach adopted by MEDHA: Length-Aware Relative Slack (LARS).

Problem Formulation. We consider a stream of requests $\mathcal{R} = \{r_1, r_2, ...\}$ where each request r_i is characterized by: arrival time a_i, total work requirement w_i^{total} (total computation time), and deadline d_i relative to its arrival. At any time t, a request has remaining work $w_i(t)$ where $w_i(a_i) = w_i^{\text{total}}$. The scheduler must assign each request to time slots on available GPUs to maximize goodput – the fraction of requests meeting their deadlines, without introducing any systematic bias towards long or short requests.

Illustrative Scenario. To understand how different policies handle heterogeneous workloads, consider a simple instance with three requests. Let request r_L have $w_L^{\text{total}} = 10$ seconds with deadline $d_L = 16$ seconds. Let requests r_S^1, r_S^2 each have $w_S^{\text{total}} = 0.5$ seconds with deadlines $d_S = 1$ second. Request r_L arrives at time 0; r_S^1, r_S^2 arrive at time 5. How should we schedule these requests to meet their deadlines?

Straw-man Solutions. Most inference systems default to First-Come, First-Served (FCFS) for its simplicity and fairness [2, 6, 19, 31, 42]. Under FCFS, r_L executes from time 0 to 10. Requests r_S^1, r_S^2 wait until time 10, missing their deadlines at time 6. This demonstrates the convoy effect: short requests experience deadline violations when queued behind long-running requests. The non-preemptive nature that makes FCFS simple also makes it unsuitable for heterogeneous SLO requirements. The convoy effect is a fundamental problem in scheduling heterogeneous workloads.

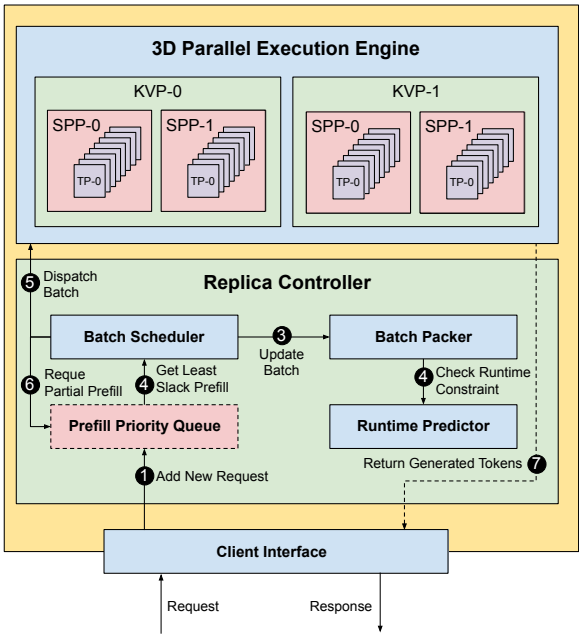


Figure 8: MEDHA architecture for efficient long-context inference. The *Replica Controller* centrally manages the request life-cycle, featuring a slack-aware *Batch Scheduler* and a *Batch Packer* to optimize for SLOs. It dispatches batches to the *3D Parallel Execution Engine*, which leverages MEDHA's novel combination of KVP, SPP, and TP.

The natural solution is to prioritize urgent requests. Earliest Deadline First (EDF) implements this intuition directly: always schedule the request whose deadline is soonest. Initially, EDF works as intended. At time 5, r_S^1, r_S^2 preempt r_L since $d_S^1 = d_S^2 = 6 < d_L = 16$. However, continuous arrivals of short requests cause r_L to accumulate delay. Once time exceeds d_L, the system enters a pathological state: r_L now has a deadline $d_L <$ current time t, necessarily earlier than any future arrival. EDF then executes r_L to completion, creating a convoy. The policy exhibits two distinct failure modes: initial starvation of long requests, followed by convoy formation for short ones. Under a constant stream of requests, both long and short requests end up missing their deadlines, and the EDF ends up performing comparable to the non-preemptive FCFS baseline (Section 6.4).

Length-Aware Relative Slack (LARS). Slack-based scheduling naturally captures deadline urgency by tracking the time buffer before violation: $s_i = a_i + d_i - t - w_i(t)$. However, in heterogeneous workloads, raw slack values can be mislead-

ing. Two requests with identical 2-second slack face vastly different risks if one requires 5 seconds of work while another requires 100 seconds – the longer request must survive far more scheduling decisions and potential preemptions.

LARS refines slack-based scheduling for heterogeneous workloads by scaling slack relative to work requirement: $\rho_i = s_i / w_i^{\text{total}}$. In our example, when r_L arrives with absolute slack of 6 seconds but relative slack $\rho_L = 0.6$, LARS recognizes its vulnerability – despite the seemingly comfortable buffer, it has limited slack per unit of work. Short requests arriving with $\rho_S = 1.0$ can afford to wait initially; they preempt only when their relative slack drops below the long request's.

This approach ensures all requests, regardless of length, make proportional progress toward their deadlines. Short requests still receive priority when urgent (low relative slack), but long requests aren't perpetually starved as in EDF. The result is a natural balance that avoids both convoy effects and starvation.

4.2 Multi-Prefill Batching by Exploiting Arithmetic Intensity Slack

Beyond preemptive scheduling, MEDHA improves system throughput by optimizing the composition of each batch. In standard chunked prefill-based scheduling [6], typically only one prefill at a time. This is based on the rationale that batching multiple prefills does not improve throughput, because a single prefill chunk is already sufficient to saturate the GPU compute. However, with adaptive chunking, we observe that prefill operations at different stages of their execution have complementary resource needs – and can benefit from batching.

Arithmetic Intensity Slack in Adaptive Chunking. We observe an opportunity to piggyback computation of short prefill requests with long prefills that allows us to compute the short prefills at a negligible cost. As Section 3.2, the adaptive chunking policy dictates that we must use smaller chunks for later stages of long context prefills (when the attention cost is dominant). While the small chunk size is sufficient to saturate the attention operation, it leaves the MLP operation memory-bound. This creates ***arithmetic intensity slack***, *i.e.,* unused computational capacity within a batch – which can be used to perform additional compute for negligible cost.

Multi-Prefill Batching Policy. To leverage this slack, the scheduler co-locates two prefill chunks in the same batch with complementary profiles – packing a short, early-stage (MLP-dominant) chunk and one long, late-stage (Attention-dominant) chunk. As shown in Section 6.4, results in 1.8× improvement in overall system throughput.

4.3 Dynamic Batch Packing for SLO Adherence

Time Budget. The batch packer constructs batches that maximize throughput while respecting a strict iteration time budget t_{target}. This budget, derived from decode requests' TPOT SLOs, ensures prefill operations cannot delay latency-sensitive decode tokens. Every scheduling cycle must complete within t_{target} to maintain predictable decode latencies.

Iterative Batch Packing. The batch packer employs a two-phase greedy algorithm guided by a runtime performance model Algorithm 1. First, it adds all active decode requests, establishing a baseline execution time t_{decode}. Second, it iteratively fills the remaining budget ($t_{target} - t_{decode}$) with prefill chunks in LARS priority order. For each prefill, the packer uses binary search to find the maximum chunk size that fits within the remaining budget. This process continues until the budget is exhausted or no viable chunks remain.

Critically, this fixed-budget approach naturally implements adaptive chunking (Section 3.2): early-phase prefills with empty KV caches fit large chunks within the budget, while late-phase prefills with populated KV caches are constrained to smaller chunks.

Space Sharing for Multi-Prefill Batching. As discussed in Section 4.2, the packer co-locates prefill chunks from different requests to fill the arithmetic slack and improve system throughput. Long prefills voluntarily yield a portion of their time budget based on their slack – a request with relative slack ρ uses only $(1-\rho) \times t_{target}$, capped at a maximum yielding fraction. This mechanism creates space for short prefills without jeopardizing the long request's deadline.

Consider a long prefill with 20% relative slack: it yields 20% of the budget, using 16ms of the 20ms allocation. The remaining 4ms allows the packer to insert short prefills, improving overall throughput while both requests progress toward their deadlines. To prevent contention, at most one long prefill is scheduled per batch.

Runtime Prediction. The packer relies on accurate runtime predictions to make informed decisions. We use Vidur [5], a performance model with <5% prediction error. The model accounts for both chunk size and KV cache state, enabling the packer to precisely fill the time budget without violations. Through binary search over possible chunk sizes, the packer maximizes resource utilization within each scheduling cycle.

5 Implementation

MEDHA extends the Sarathi-Serve framework [6] 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 [14] with ZeroMQ [3] for scheduler-worker communication, eliminating GIL contention as we scale to hundreds of workers. We also integrate FlashInfer [46] 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.

6 Evaluation

6.1 Evaluation Setup

Baselines. We compare our system against the state-of-the-art long-context LLM inference serving systems, LoongServe [42] and vLLM [19]. Note that, for context lengths greater than 32K, vLLM defaults to the Sarathi-Serve scheduler [6]. Thus, we refer to this baseline as Sarathi. We consider two chunk sizes for the Sarathi scheduler: 512 and 2048. We also consider DistServe [49] and SplitWise [29], however, these systems run out of memory due to activation memory bottleneck as discussed in Section 2.2. To our knowledge, there are no publicly available systems that directly tackle convoy effects in long context inference. Finally, we evaluate MEDHA variant that replaces the LARS request prioritization and multi-prefill batching with standard FCFS/EDF/LRS scheduling while retaining all other proposed mechanisms.

Models and datasets. We use Llama-3 8B and Llama-3 70B with RoPE [38] 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 [8] and LV-Eval [48] for long context evaluations. These datasets were created to evaluate long-context abilities of LLMs and predominantly contain short form question, with extremely small decode lengths. For instance, the median output length in L-Eval is 47 tokens as opposed to 415 in ShareGPT4 [41] — which is based on actual real-world user interactions with GPT4.

To perform more realistic evaluations, we construct the **MEDHA-*SWE*** trace using the Gemini-Flash-1.5B model [34], inspired by LLM-enabled software engineering tools that have recently gained popularity. We focus on two common engineering tasks: code review for pull requests and GitHub issue resolution. 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 solve them referencing the 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 [41], 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 [26]. While for Llama-3 70B, we use a 128-GPU cluster with 16 DGX-H100 servers [25]. 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.

6.2 Capacity Evaluation

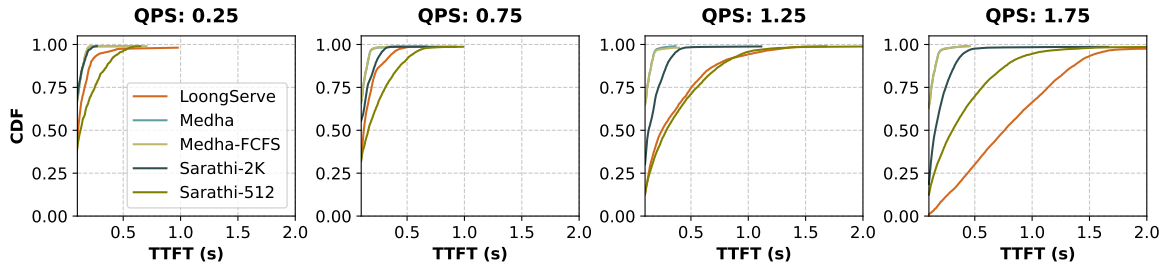
We begin by evaluating how MEDHA performs under varying loads compared to existing approaches for Llama-3 8B model on the A100 cluster. Our capacity evaluation focuses on two key metrics: TTFT and TPOT, as these directly impact user experience in interactive scenarios.

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).

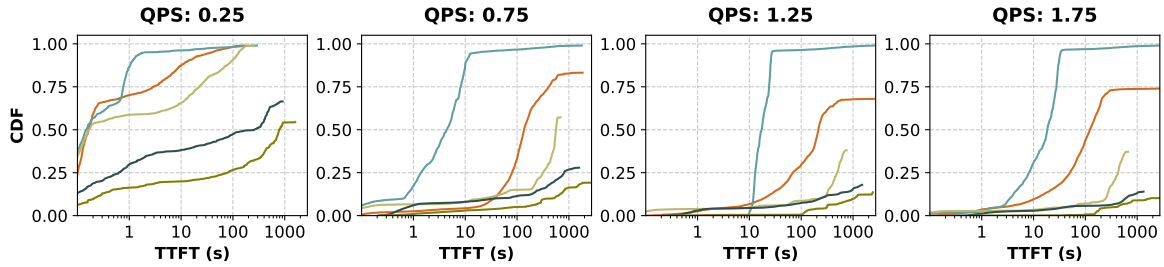
Baseline Performance. In the scenario with only short requests (Figure 9a), all systems exhibit comparable performance at low loads (0.25 QPS). However, as load increases, LoongServe's performance degrades considerably, which we attribute to resource fragmentation. At 1.75 QPS, LoongServe's P90 TTFT increases dramatically, while MEDHA maintains consistent latency. Furthermore, MEDHA achieves considerably better latency compared to Sarathi due to MEDHA's SPP, which helps reduce TTFT.

Long Query Performance. Figure 9b 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 convoy effect. 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 convoy effect, resulting in truncated CDFs.

Decode Performance. Figure 10 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 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

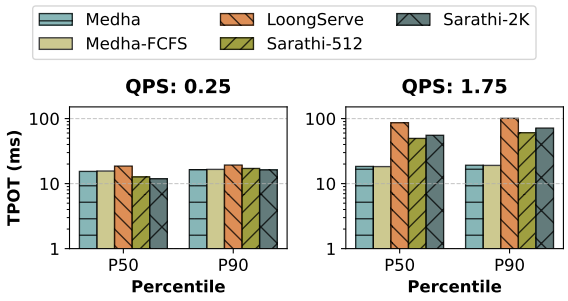


(a) For short-context workloads from ShareGPT4, MEDHA maintains consistently low latency even at high QPS.

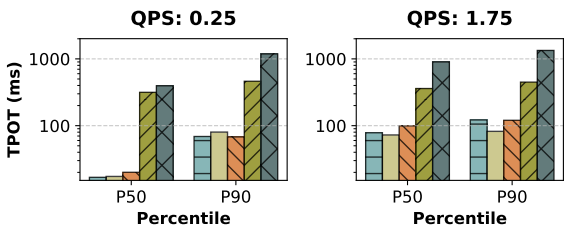


(b) For ShareGPT4 with 5% long requests, MEDHA achieves up to 30× lower median TTFT, demonstrating effective mitigation of HOL blocking

Figure 9: 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 10: 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.

adaptive chunking maintains consistent performance across varying sequence lengths.

6.3 3D Parallel Performance

With MEDHA's baseline established, we evaluate 3D parallelism that combines tensor, stream pipeline and KV parallelism. For this experiment we use Llama-3 70B on a 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. We run a mixed workload, including 5% long-context (2M token) requests, scaled from the MEDHA-SWE trace.

Figure 11 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 the higher SPP degree in the 2D case, which is more communication-efficient than KVP and better accelerates prefill. Despite this, both configurations maintain similar median latencies.

Figure 12 shows the strength of 3D parallel in the decode phase. At high load (1.75 QPS), the 3D 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, 3D parallelism delivers more consistent end-to-end latency—critical for real-

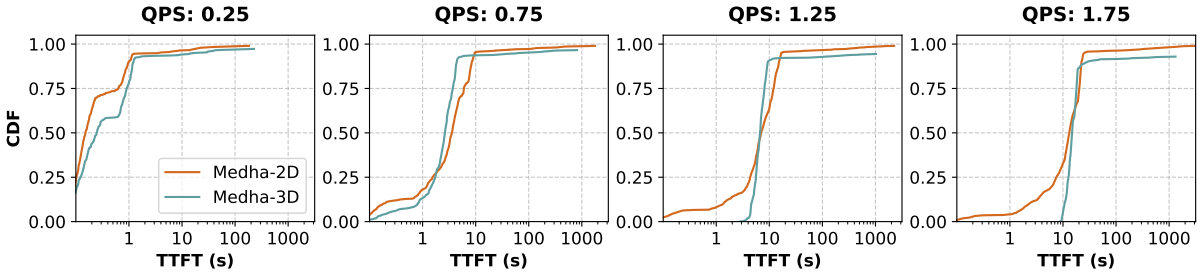


Figure 11: Prefill performance comparison of parallelization strategies for Llama-3 70B on 8 64 H100 GPUs running ShareGPT4 with 5% long requests.

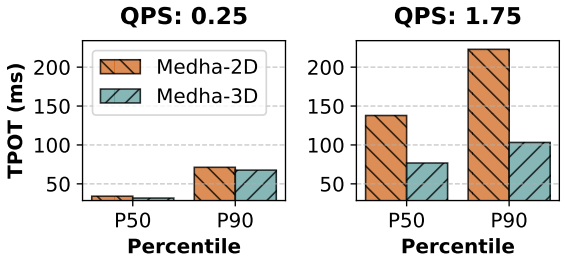


Figure 12: MEDHA-3D (SPP+TP+KVP) maintain comparable TTFT performance MEDHA-2D (SPP+TP) and but enable 2× better decode performance by distributing KV cache reads and reducing prefill-decode interference.

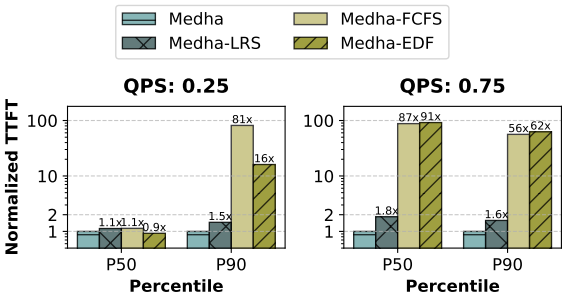


Figure 13: Impact of different scheduling policies on normalized TTFT latency. Even compared to our modified LRS policy, MEDHA scheduler achieves (1.6–1.8×) lower latency, demonstrating the effectiveness of MEDHA's prefill-prefill batching technique.

world deployments. It retains the benefits of SPP while combining the strengths of both approaches.

6.4 Effectiveness of MEDHA Scheduler

We isolate the performance gains from MEDHA's scheduling policies by comparing it to traditional scheduling policies. Figure 13 shows the TTFT distributions for four approaches: FCFS, EDF, LARS (without multi-prefill batching), and MEDHA's scheduler with all optimizations enabled. The evaluation uses Llama-3 8B on A100 GPUs in TP8-SPP2 configuration with a mixed workload of 5% long-context requests.

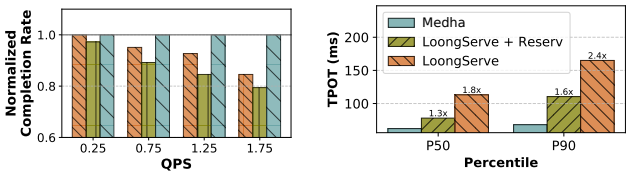
At low load (0.25 QPS), all policies show similar median latency but differ in tail behavior. However, at high load (1.75 QPS), the differences become more pronounced. FCFS performs poorly due to unmitigated convoy effect from long requests. Despite its success in latency-sensitive systems, EDF struggles 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 gain highest priority once they pass their deadlines as discussed in Section 4.1.

We also compare MEDHA with multi-prefill batching to vanilla LARS. While both of these setups significantly outperform the FCFS and EDF baselines by mitigating convoy effect, we up to 1.8× lower median latency with MEDHA compared to the vanialla LARS setup due to more effective GPU utilization enabled by multi-prefill batching.

Sensitivity to Long Request Mix. Figure 1a shows how TTFT degrades as the fraction of long requests increases from 0% to 5%. The baseline system exhibits superlinear degradation — even with 1% long requests LoongServe shows 8× higher latency, while at 5% it exhibits 30× P50 and 174× P90 higher latency due to convoy effect. In contrast, MEDHA gracefully maintains the P90 latency under 10 seconds even with 5% long request mix.

6.5 Alternate Scheduling Approaches: Multiuple Request Pools

A common industry technique to tackle with heterogeneity 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 14.



(a) Normalized completion rate. (b) Decode latency at 0.75 QPS.

Figure 14: 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.

We compare MEDHA to this baseline using the same setup as Section 6.2, 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. Thus, creating separate pools for requests of different length does not solve the fundamental problem of convoy effect while hurting throughput due to fragmentaion.

7 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 [11, 20, 22]. We discuss and compare them in detail in Sections 2.1 and 6. A similar idea to SPP, called token-parallelism, was used in TeraPipe [21] 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. Recently, Mooncake [31] –Kimi.ai's proprietary serving system, a work parallel to ours, concurrently proposed use of this technique to reduce TTFT latency during inference. Note that, while Mooncake explores use of chunked prefills to accelerate long prefill computation, it does not address convoy effect. To the best of our knowledge, MEDHA is the first system to leverage chunked prefills for preemptive scheduling to tackle heterogeneity in long context serving.

Request scheduling. Efficient request scheduling has been extensively studied [15, 23, 32, 33, 35, 39, 43], but existing approaches have notable limitations when addressing long-context requests. For example, SRTF scheduling [15, 33] reduces median latency but leads to starvation of long requests due to lack of preemption. LoongServe [42] 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 [35] 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.

8 Conclusion

This work demonstrates that the convoy effect, long understood in operating systems, is a critical but overlooked challenge in long-context LLM serving. Traditional non-preemptive systems fail to tackle the extreme heterogeneity caused by the quadratic attention cost, as a result a single long request can drastically degrade service for hundreds of short queries. Our results show that, that with careful co-design of parallelism strategies and scheduling policies preemption can be both practical and effective. As context windows of state-of-the-art LLMs continues to grow, the heterogeneity problem will only intensify, making preemptive scheduling a requirement rather than an optimization. MEDHA shows that with careful system design, we can effectively serve long-context LLM workloads at scale.

References

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

[2] TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference. https://github.com/NVIDIA/TensorRT-LLM.

[3] ZeroMQ. https://zeromq.org/.

[4] Amey Agrawal, Anmol Agarwal, Nitin Kedia, Jayashree Mohan, Souvik Kundu, Nipun Kwatra, Ramachandran Ramjee, and Alexey Tumanov. Etalon: Holistic Performance Evaluation Framework for LLM Inference Systems, 2024.

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

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

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

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

[9] Anthropic. Claude code: An agentic coding tool for terminal-based development. https://github.com/anthropics/claude-code, 2025. GitHub repository.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2023. Chapter: CPU Scheduling.

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

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

[13] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning, 2023.

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

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

[16] Google. Gemini – Long context, 2024.

[17] Google Gemini. Gemini cli: An open-source ai agent for terminal workflows. https://github.com/google-gemini/gemini-cli, 2025. GitHub repository.

[18] 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.

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

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

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

[22] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring Attention with Blockwise Transformers for Near-Infinite Context, 2023.

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

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

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

[26] Microsoft Azure. NDm-A100-v4 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndma100v4-series?tabs=sizebasic, 2024.

[27] Tiyasa Mitra, Ritika Borkar, Nidhi Bhatia, Ramon Matas, Shivam Raj, Dheevatsa Mudigere, Ritchie Zhao, Maximilian Golub, Arpan Dutta, Sailaja Madduri, et al. Beyond the buzz: A pragmatic take on inference disaggregation. *arXiv preprint arXiv:2506.05508*, 2025.

[28] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–15, 2021.

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

[30] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. POLCA: Power Oversubscription in LLM Cloud Providers, 2023.

[31] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, and Xinran Xu Weimin Zheng. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving, 2024.

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

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

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

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

[36] 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.

[37] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 9th edition, 2012.

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

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

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

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

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

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

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

[45] 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. Qwen2.5-1m technical report. *arXiv preprint arXiv:2501.15383*, 2025.

[46] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. Accelerating Self-Attentions for LLM Serving with FlashInfer, February 2024.

[47] 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 *OSDI*, 2022.

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

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

A Scheduling Algorithms

This appendix presents the detailed algorithms used in MEDHA's scheduling and batch packing mechanisms. We provide pseudocode for the adaptive batching process, chunk size selection, and prefill time estimation procedures.

A.1 Batch Formation and Packing

The batch formation process operates in two phases: first packing latency-sensitive decode requests, then filling remaining capacity with adaptively-chunked prefill requests. Algorithm 1 presents the core batching logic.

Algorithm 1 SLO-Aware Adaptive Batching

Require: $\mathcal{Q}$: Queue of pending requests (prefill and decode)
Require: t_{target}: Target batch execution time (e.g., 20ms from TPOT SLO)
Ensure: $\mathcal{B}$: Batch of (request, num_tokens) pairs

procedure FORMNEXTBATCH($\mathcal{Q}, t_{target}$)
  $\mathcal{B} \leftarrow \emptyset$ ▷ Batch of (request, tokens) pairs
  $t_{pred} \leftarrow 0$ ▷ Predicted batch execution time
  Phase 1: Pack decode requests
  $\mathcal{D} \leftarrow$ GetDecodeRequests($\mathcal{Q}$)
  $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{D}$
  $t_{pred} \leftarrow$ PredictTime($\mathcal{B}$)
  Phase 2: Fill with prefill chunks
  $\mathcal{P} \leftarrow$ GetPrioritizedPrefills($\mathcal{Q}$) ▷ LARS-ordered
  while $\mathcal{P} \neq \emptyset$ and $t_{pred} < t_{target}$ **do**
    $r \leftarrow$ Pop($\mathcal{P}$)
    $n \leftarrow$ GetChunkSize($r, t_{pred}, t_{target}, \mathcal{B}$)
    if $n > 0$ **then**
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{(r, n)\}$
      $t_{pred} \leftarrow$ PredictTime($\mathcal{B}$)
    else
      Requeue($r, \mathcal{P}$)
    end if
  end while
  return $\mathcal{B}$
end procedure

The algorithm prioritizes decode requests to meet their strict TPOT requirements, then iteratively adds prefill chunks based on LARS priority ordering. The target batch time t_{target} serves as a hard constraint derived from decode SLOs.

A.2 Adaptive Chunk Size Selection

The chunk size selection mechanism implements both the adaptive chunking policy and space-sharing optimization. Algorithm 2 shows how chunk sizes are determined based on current batch composition and request urgency.

The algorithm implements two key policies: (1) space-sharing where requests with high relative slack ρ yield time to more urgent requests, and (2) finding the maximum chunk size that fits within the allocated time budget.

Algorithm 2 Adaptive Chunk Size Selection

Require: r: Prefill request with KV cache size $r.kv\_size$
Require: $t_{current}$: Current predicted batch execution time
Require: t_{target}: Target batch execution time limit
Require: $\mathcal{B}$: Current batch composition
Require: ρ_{max}: Maximum space-sharing fraction (e.g., 0.4)
Ensure: Chunk size in tokens, or 0 if request cannot be scheduled

procedure GETCHUNKSIZE($r, t_{current}, t_{target}, \mathcal{B}$)
  Safety constraint: Prevent multiple long prefills
  if IsLong(r) $\wedge$ ContainsLongPrefill($\mathcal{B}$) **then**
    return 0
  end if
  Space sharing: Long requests yield time for urgency
  $\rho \leftarrow$ GetRelativeSlack(r) ▷ $\rho = s(t)/w^{\text{total}}$
  $\rho \leftarrow \min(\rho_{max}, \max(0, \rho))$ ▷ Cap at max sharing
  Calculate effective budget:
  $t_{effective} \leftarrow t_{target} \cdot (1 - \rho)$
  $t_{budget} \leftarrow t_{effective} - t_{current}$
  if $t_{budget} \leq 0$ **then**
    return 0
  end if
  Binary search for maximum chunk:
  return BinarySearchChunk(r, t_{budget})
end procedure

A.3 Prefill Time Estimation

Accurate estimation of remaining prefill time is crucial for LARS scheduling. We precompute a cache of prefill times for various sequence lengths, accounting for the adaptive chunking policy. Algorithm 3 presents both the offline precomputation and runtime estimation procedures.

The simulation accounts for how chunk sizes decrease as the KV cache grows, reflecting the shift from MLP-dominant to attention-dominant computation. The SIMULATEPREFILL procedure uses the Vidur simulator [5] to model execution times accurately.

B Scaling Properties of Parallelism Strategies

This appendix provides detailed scaling analysis of MEDHA's parallelism strategies: Stream Pipeline Parallelism (SPP) for prefill acceleration and KV-Cache Parallelism (KVP) for decode latency bounding.

B.1 Stream Pipeline Parallelism Scaling

Figure 17 demonstrates the scaling efficiency of Stream Pipeline Parallelism across different model sizes and sequence

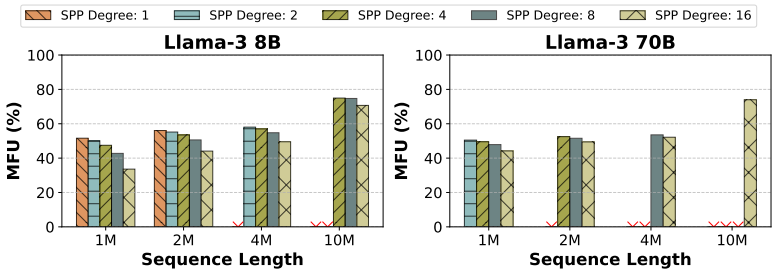


Figure 15: Model FLOPS Utilization [12] (MFU) for MEDHA 2D (TP+SPP). It achieves 50-60% utilization across sequence lengths and parallelism degrees.

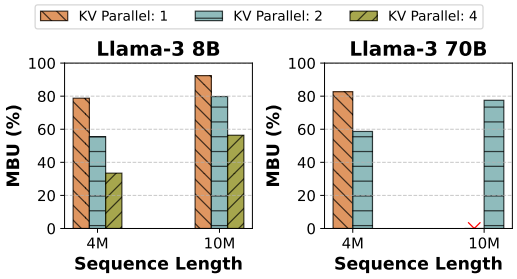
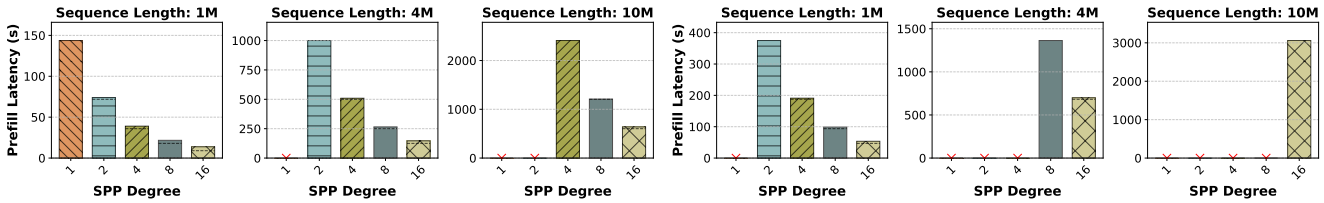


Figure 16: Model Bandwidth Utilization (MBU) for MEDHA 2D (TP+KVP).



(a) Llama-3 8B.

(b) Llama-3 70B.

Figure 17: 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.

lengths. We evaluate MEDHA 2D (SPP+TP) configurations against baseline approaches for long-context prefill processing.

For Llama-3 8B (Figure 17a), MEDHA achieves near-linear scaling up to 128 H100 GPUs. The efficiency remains above 70% even at the highest parallelism degrees, demonstrating effective overlap of computation and communication. Notably, MEDHA outperforms ring attention approaches by 60% due to eliminating the sequential dependency bottleneck.

Scaling to Llama-3 70B (Figure 17b) shows even stronger benefits. The larger model's increased compute density better amortizes pipeline startup costs, achieving 85% scaling efficiency at SPP degree 8.

Figure 18 examines the decode latency implications of SPP scaling. Due to it's communication efficient nature, SPP only marginally affects decode performance due to pipeline depth. With SPP degree 8, decode latency increases by only 16%.

B.2 KV-Cache Parallelism Impact on Decode Performance

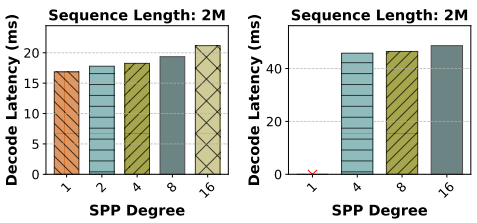
Figure 19 shows KVP's effectiveness in bounding decode latency. For 10M-token contexts on Llama-3 8B, KVP with degree 4 reduces TPOT by 40% in decode-only batches. The scaling is sub-linear due to communication overhead, but the latency reduction is crucial for meeting decode SLOs with long contexts.

B.3 TTFT-TPOT Trade-off Analysis

We sweep the space of various chunk sizes for the chunked prefill, and also vary p_{kvp}, while keeping $p_{spp} = 4$. Figure 20 shows the results on Llama-3 8B. For a given p_{kvp}, increasing the chunk size, reduces TTFT (prefill latency), since it requires fewer iterations. At the same time, it increases TBT, since each batched iteration takes longer to execute. Therefore, for sequence length 1M with $p_{kvp} = 1$, the green line shows the left-most triangle at largest chunk size, and the right-most triangle at the smallest chunk size. For a given chunk size, increasing p_{kvp} helps reduce both TTFT and TBT in most cases, thus helping reach more optimal points in this trade-off space. Indeed, lower p_{kvp} achieves better TTFT latency in cases with lower arithmetic intensity (due to small chunk size), as exemplified by the right-most points for 1M context length. As we increase the arithmetic intensity (*e.g.*,, 2M context length), we see increasing p_{kvp} achieving the same performance for the smallest chunk size, and, finally, decreasing TTFT for 4M context length.

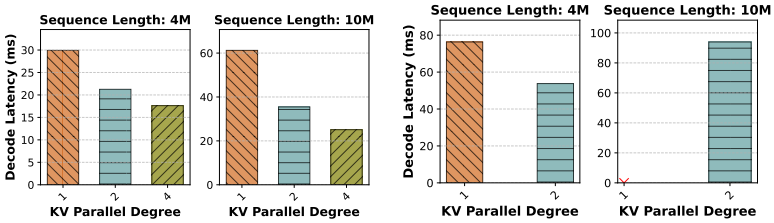
B.4 Resource Utilization Efficiency

A key 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



(a) Llama-3 8B (b) Llama-3 70B

Figure 18: 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 19: 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.

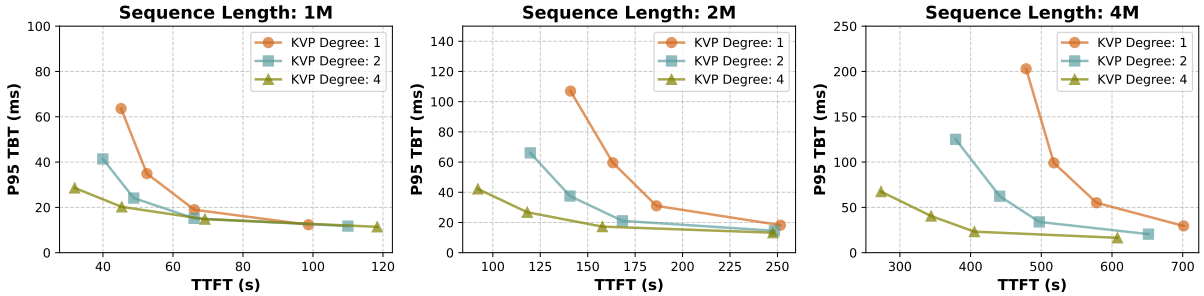


Figure 20: Trade-off Between TTFT and P95 TBT for Llama-3 8B using MEDHA 3D Parallelism ($p_{tp} = 4$, $p_{spp} = 4$) for varying KVP degrees and chunk sizes (32-256).

(MBU) [1, 12]. In LLM inference, prefill phases are compute-bound while decode phases are memory-bound [29, 30]. Figure 15 shows the MFU for MEDHA in the prefill phase (2D SPP+TP), while Figure 16 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 16 shows that MEDHA's KVP implementation achieves up to 92% MBU in optimal configurations, allowing consistent decode performance even with extremely long contexts.

Algorithm 3 Remaining Prefill Time Estimation

Offline Precomputation:
Require: L_{max}: Maximum sequence length to cache (e.g., 1M tokens)
Require: Δ: Granularity of cache entries (e.g., 1K tokens)
Ensure: $\mathcal{C}$: Cache mapping sequence length to total prefill time
procedure BUILDPREFILLCACHE(L_{max}, Δ)
    $\mathcal{C} \leftarrow \{0 \mapsto 0\}$ ▷ Cache: tokens $\rightarrow$ time
    for $\ell = \Delta$ to L_{max} step Δ **do**
        $\mathcal{C}[\ell] \leftarrow$ SimulatePrefill(ℓ)
    end for
    return $\mathcal{C}$
end procedure

Require: L: Total sequence length to simulate
Ensure: Total time to prefill L tokens with adaptive chunking
procedure SIMULATEPREFILL(L)
    $t_{total} \leftarrow 0$
    $\ell_{processed} \leftarrow 0$
    while $\ell_{processed} < L$ **do**
        $kv_{size} \leftarrow \ell_{processed}$ ▷ Current KV cache size
        $(c, t_c) \leftarrow$ GetOptimalChunk(kv_{size}, t_{target})
        if $c = 0$ **then**
            break
        end if
        $t_{total} \leftarrow t_{total} + t_c$
        $\ell_{processed} \leftarrow \ell_{processed} + c$
    end while
    return t_{total}
end procedure

Runtime Estimation:
Require: ℓ_{total}: Total tokens in the request
Require: $\ell_{processed}$: Tokens already processed
Ensure: Estimated time to complete remaining tokens
procedure GETREMAININGTIME($\ell_{total}, \ell_{processed}$)
    $t_{total} \leftarrow$ LookupCache($\mathcal{C}, \ell_{total}$)
    $t_{done} \leftarrow$ LookupCache($\mathcal{C}, \ell_{processed}$)
    return $t_{total} - t_{done}$
end procedure