Optimizing Long-context LLM Serving via Fine-grained Sequence Parallelism

Cong Li[*1] Yuzhe Yang[2] Xuegui Zheng[2] Qifan Yang[2] Yijin Guan[3] Size Zheng[2]
Li-Wen Chang[2] Shufan Liu[2] Xin Liu[2] Guangyu Sun[1]

[1]*Peking University* [2]*Bytedance Seed* [3]*Bytedance*

Abstract

With the advancement of large language models (LLMs), their context windows have rapidly expanded. To meet diverse demands from varying-length requests in online services, existing state-of-the-art systems tune the sequence parallelism (SP) allocation. However, current dynamic SP allocation lacks flexibility to (1) support stage-specific parallelism requirements in LLM inference, (2) mitigate the global latency degradation from excessive SP allocation, and (3) exploit resource fragments arising from SP size variation.

To tackle this problem, we propose Chunkwise Dynamic Sequence Parallelism (CDSP), a fine-grained parallelism strategy that assigns SP sizes across *intra-request* token segments. Based on CDSP, we build Tetris, an LLM serving system that (1) efficiently integrates CDSP into disaggregated cluster to satisfy parallelism heterogeneity, (2) dynamically regulates SP size expansion based on real-time load conditions, and (3) adaptively explores chunking plans to utilize fragmented resources while meeting per-request demands. Compared with state-of-the-art systems, Tetris achieves up to 4.35× lower time-to-first-token (TTFT) under max sustainable loads, reduces median time-between-tokens (TBT) by up to 40.1%, and increases the max request capacity by up to 45%.

1 Introduction

Large Language Models (LLMs) have empowered many generative tasks such as chatbot [12, 28], code completion [11, 24], and reasoning [40, 41]. Such capability drives many cloud companies to deploy online LLM services [2, 4, 12, 28]. As LLMs continue to advance, their context lengths have notebly expanded. For example, OpenAI's GPT-4o [29] supports 128K contexts, Anthropic's Claude-3 [3] supports 200K, and Google's Gemini-2.5 pro [13] supports up to 1M tokens.

With the growth of sequence length, LLM inference requires proportionally more resources. To augment resource provision for long-context requests, sequence parallelism (SP) has been widely applied [5, 10, 15–17, 19, 20, 39, 42, 43]. Among these implementations, ring-attention-based SP [20] (also known as context parallelism [10, 39, 43]) has been introduced to LLM serving [42, 43]. Specifically, it scatters long sequences across multiple LLM instances and performs distributed attention computation through peer-to-peer (P2P) KV cache transmission. By overlapping cache transmission with attention computation, ring attention demonstrates better scalability than tensor parallelism (TP), especially when populating resources beyond a single node [43].

The expansion of context window also widens request length gaps, thereby amplifying variability in per-request resource demands. To cope with this, existing state-of-the-art long-context LLM serving system, LoongServe [42], proposes elastic sequence parallelism (ESP). ESP dynamically adjusts SP allocation *in the granularity of request batch* to satisfy diverse resource demands. In contrast, non-SP systems have to statically configure resource allocation at startup due to the high overhead of model weight resharding, limiting their ability to respond to highly variable resource demands when serving long-context LLMs.

Although LoongServe has surpassed existing best-performing non-SP systems [1, 18, 22, 46], its *coarse-grained SP allocation* fails to fully optimize online long-context LLM serving's performance: First, ESP enforces a uniform TP size across all instances. However, prefill benefits from smaller TP for better resource allocation flexibility, while decoding prefers larger TP to minimize compute latency. Second, LoongServe assigns requests to fixed batches and exhaustively optimizes per-batch latency. However, since this local-optimal strategy lacks global load awareness, its excessive SP expansion fails to optimize system's overall latency distribution. Third, dynamic SP allocation leads to varying queuing delays across instances. However, since ring attention requires synchronous computation across instances, such an imbalance results in idle slots and degrades overall resource efficiency.

To tackle these problems, we first propose Chunkwise Dynamic Sequence Parallelism (CDSP), a *fine-grained intra-request SP allocation* strategy. It splits each request's prompt into multiple chunks and assigns each chunk a distinct SP size, enabling efficient utilization of resource fragments while fully optimizing prefill latency. Based on CDSP, we build Tetris, a system for efficient online long-context LLM serving. Tetris efficiently integrates CDSP into prefill-decoding disaggregated cluster by extending attention load-balancing strategy and KV cache transfer management, thereby fully accommodating the parallelism heterogeneity across different stages. For online scheduling, Tetris regulates SP size allocation based on real-time request arrival pressure to prevent excessive SP expansion from degrading global latency. In addition, Tetris integrates a load-aware chunk partitioning

[*] Work done during Cong Li's internship at Bytedance Seed.

scheme that dynamically determines the optimal execution plan for each request, maximizing the benefits of CDSP. To summarize, we have made the following contributions:

- We identify existing dynamic SP allocation strategy's rigidity in handling inter-request resource variability under online long-context LLM serving scenarios.
- We propose CDSP for *intra-request fine-grained* SP allocation and build Tetris's inference engine to fully satisfy the heterogeneous demands in long-context LLM serving.
- We propose *real-time load-aware* SP size allocation and chunk partitioning strategies in Tetris's scheduler to optimize the service's overall latency distribution.

Extensive experiments on workloads collected from a *real-world online long-context LLM service* demonstrate that Tetris achieves up to 4.35× lower time-to-first-token (TTFT) under state-of-the-art systems' max sustainable loads, reduces median time-between-tokens (TBT) by up to 40.1%, and increases the max request capacity by up to 45%.

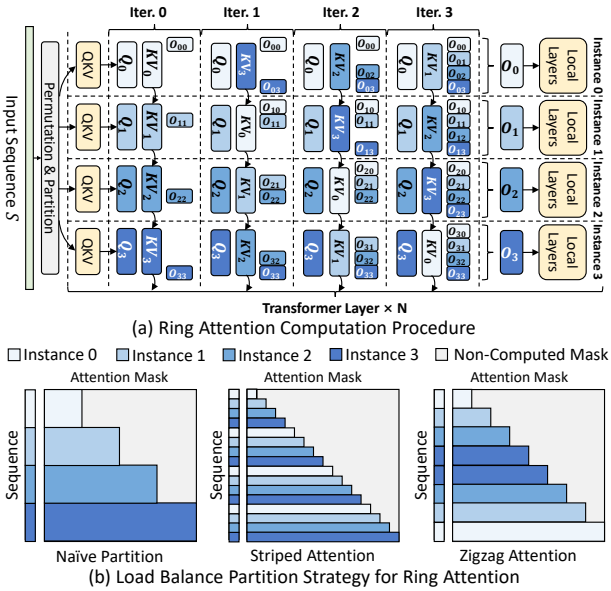

Figure 1. Ring-Attention-Style Sequence Parallelism.

2 Background and Motivation

2.1 Transformer-based LLMs

Mainstream LLMs are built on transformer decoder layers [38], which contain an attention block and a feed-forward network (FFN) block. In the attention block, the inputs are projected to query, key, and value vectors, which interact with each other through self-attention. Then, the outputs of the attention block are processed by multi-layer perceptrons (MLPs) in the FFN block to produce the decoder layer outputs. After passing a stack of transformer layers, the final outputs can be used for downstream generative tasks.

LLM's generation procedure contains two stages: prefill and decoding. In the prefill stage, the LLM processes all tokens of the input prompt in parallel to produce the first token. Then, in the decoding stage, the LLM takes the previous token as input and predicts one new token per iteration, gradually building the full output sequence. Since self-attention requires each token to interact with all previous tokens' key/value vectors, these intermediate states are stored throughout LLM inference to avoid redundant computation, which is known as KV Cache [32].

2.2 LLM Serving

Online LLM service has been widely deployed by cloud companies [2, 4, 12, 28], which receives requests from multiple users, conducts inference on a GPU cluster, and returns decoding outputs in real-time. To evaluate the serving quality (or Service Level Objectives, SLOs), service providers proposed several metrics: The Prefill stage is measured by time to first token (TTFT), which is the duration between request arrival and the finish of prefill computation. For decoding stage, time between tokens (TBT) is employed to measure the smoothness of the output streaming procedure.

To optimize these SLOs and improve the serving system's efficiency, several system optimizations have been proposed: Iteration-level scheduling adds new requests once the current decoding iteration finishes, reducing the queuing latency of each request [44]. PagedAttention eliminates the memory fragmentation caused by the variance of prompt and decoding lengths via managing the KV cache in block granularity [18]. Prefill-decoding disaggregation routes requests under different stages to distinct model instances to avoid the interference between the two stages [46].

2.3 Sequence Parallelism for Long-Context LLMs

Sequence parallelism (SP) has been a pivotal approach to handle long-context requests' compute and memory demands [5, 10, 15–17, 19, 20, 39, 42, 43]. In this paper, we mainly focus on ring-attention-style SP, which has been adopted in LLM inference [42, 43]. As shown in Fig. 1-(a), ring attention distributes the tokens of one sequence to multiple model instances. During the prefill stage, each instance first calculates its local tokens' query, key, and value tensors together with their attention results. Then, it sends key-value tensors to the next neighbor and receives new key-value tensors from the previous neighbor iteratively to interact local query tensors with full key-value tensors. After the distributed attention computation, each instance computes the remaining operators without communication. During the decoding stage, instead of passing key-value tensors, ring attention transfers query vectors because their smaller data volume can reduce the ring communication overhead.

Since the causal mask adopted by LLMs only requires each token to compute with all preceding tokens, splitting the

Table 1. Prefill latency (s) comparison of LLaMA3-8B, tested on A100 GPUs. The optimal latency is marked in bold.

Prompt Length	4k	8k	16k	32k	64k	128k	256k
SP=1 Latency	0.28	0.57	1.29	3.22	9.05	29.20	OOM
SP=2 Latency	0.16	0.31	0.69	1.67	4.61	14.30	50.07
SP=4 Latency	**0.13**	**0.20**	0.39	0.92	2.43	7.32	24.77
SP=8 Latency	0.21	0.24	**0.31**	0.58	1.37	3.96	12.81
SP=16 Latency	0.39	0.43	0.46	**0.53**	**0.96**	**2.31**	**7.02**

sequence into multiple consecutive shards will lead to imbalanced workload distribution across instances, as shown in Fig. 1-(b). Several optimized partition strategies have been proposed to alleviate this issue: Striped Attention [5] partitions the sequence into evenly-spaced stripes and assigns them to each instance in a round-robin manner, so that each instance can conduct computation to every KV cache shard. Another strategy [10, 15, 43] interleaves the KV Cache across instances in a "zigzag" manner, which partitions the sequence into $2N$ shards $S_0, ..., S_{2N-1}$ for N SP instances, and allocates (S_i, S_{2N-i-1}) to instance i, In this way, each instance is assigned with identical computation workload.

2.4 Limitations of Existing SP-Serving Systems

Despite SP's strong performance, existing systems still exhibit several limitations, preventing them from fully utilizing SP in online long-context LLM serving scenarios:

Limitation #1 (Fixed-SP System): Partitioning the cluster with a fixed SP size fails to meet the inter-request resource demand variation, which manifests in two aspects: *(1) Large SP Size is an overkill for short requests*. First, excessive SP size allocation leaves each instance with only a marginal compute workload, leading to low GPU utilization. Second, the undersized compute workload cannot fully overlap ring communication, which can even cause the performance to be inferior to a reduced SP size. *(2) Small SP Size severely prolongs long requests' prefill latency*, which can even reach to tens of seconds, thereby severely hurting the system's overall TTFT distribution.

To elucidate such disparity, we benchmark the prefill latency of LLaMA3-8B [14] on A100 GPUs. Detailed setups are listed in Sec. 7.1. We set the batch size to 1 and vary the prompt length from 4k to 256k. The SP size is adjusted from 1 to 16, with the TP size of 1. As listed in Table 1, *for short lengths* (e.g., 4k, 8k), adopting a moderate SP size is enough to achieve the optimal performance. Further enlarging the SP size incurs **1.2×-3× higher latency**. *For long requests* (e.g., 128k, 256k), enlarging the SP size delivers a quasi-linear improvement, **with a latency gap of up to 43.05s**. This phenomenon remains consistent across varying TP sizes and model scales. Considering online serving processes highly dynamic requests with substantial context length variation

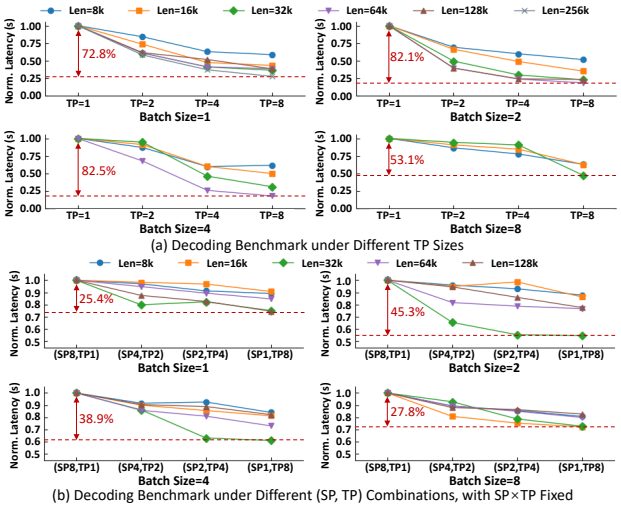

Figure 2. Decoding Latency Analysis.

as listed above, a fixed SP configuration cannot fully satisfy such diverse resource demands.

Limitation #2 (Existing Dynamic-SP System): A recent work, LoongServe [42], shares similar insights, which proposes Elastic Sequence Parallelism (ESP) to adjust resource allocation: ESP groups all instances into a unified SP pool sharing the same TP size. By assigning different SP sizes to request batches, it changes resource allocation without re-partitioning LLM parameters. Although it has achieved SOTA performance compared with best-performing non-SP systems [1, 18, 22, 46], **its inflexible SP management fails to fully unlock SP's performance benefits**, with limitations evident in three aspects:

(1) Cluster Architecture: Unified TP size fails to satisfy the disparate characteristics between prefill and decoding. **Given the device budget, larger SP size (+ smaller TP size) is preferred by prefill** in existing SP-based inference systems [42, 43] due to the following reasons: (1) SP provides more flexibility in adjusting resource provision, since we only need to split tokens across model instances. In contrast, adjusting TP requires resharding LLM's weight matrices, which suspends the underlying devices to serve new requests. (2) Compared with TP, SP demonstrates better cross-node scalability because TP's all-reduce latency increases significantly given the low inter-host network bandwidth [43]. However, **constraining decoding to prefill's small TP, as in ESP, severely degrades its performance**. To demonstrate this issue, we evaluate the decoding latency of LLaMA3-8B under different TP sizes using A100 GPUs. As shown in Fig. 2-(a), compared with TP=8, TP=1, TP=2, and TP=4 incurs up to 5.73×, 3.87×, and 1.93× higher latency, respectively. Such a slowdown severely hurts the SLO attainment of online LLM services with stringent TBT objectives [34, 46].

LoongServe mitigates this issue by augmenting decoding batches' SP size when it detects heightened resource demand.

However, given the same device budget, **increasing SP is less effective than enlarging TP for decoding**. We conduct experiments on LLaMA3-8B with 8 A100 GPUs to reveal the performance gap. As shwon in Fig. 2-(b), adopting (SP8, TP1), (SP4, TP2), and (SP2, TP4) inflates decoding latency by up to 1.83×, 1.41×, and 1.15×, respectively, relative to (SP1, TP8). Such behavior persists when larger models are partitioned across multiple GPU nodes. For example, Yang et al. [43] report that (SP2, TP8) incurs higher decoding latency than (SP1, TP16) on LLaMA3-405B. The main reason is that the scant compute workload of decoding attention is insufficient to fully mask the ring communication overhead. Therefore, an ideal online serving system should be aware of the disparity in parallelism strategy requirements to sufficiently optimize both TTFT and TBT.

(2) Batching Strategy: Greedily expanding SP size for fixed batches fails to optimize global latency distribution. LoongServe adopts **greedy static batching** for request scheduling: It selects multiple pending requests and adopts dynamic programming to decide prefill SP instances, which assigns the largest SP size to exhaustively minimize per-batch prefill latency. Once all requests finish prefill computation, the entire batch proceeds to decoding collectively. During the entire decoding stage, the batch is fixed — no additional requests are added until the phase terminates.

Batching multiple long-context requests improves the prefill throughput, which is advantageous for offline inference tasks operating on a large, pre-specified input set (e.g., post-training model evaluation). However, **combining long-context requests into one prefill batch severely hurts the system's TTFT**, as early-arriving requests have to wait for the entire batch to complete time-consuming prefill computation. Such inter-request TTFT interference should be avoided by the online service scheduler (e.g., constraining each prefill batch to a single request [34]).

Besides, **the local optimum provided by LoongServe scheduler lacks awareness of real-time load conditions**, failing to optimize the overall TTFT distribution. For example, consider a system with 16 LLaMA3-8B SP instances (TP=1), each with 1-second queuing delay. If a 32k request is greedily assigned SP=16 by LoongServe scheduler (based on Table 1), and a subsequent 16k request arrives, the TTFTs of (32k, 16k) requests are (1.53s, 1.84s). In contrast, if we assign SP=8 to the 32k request and reserve 8 instances for the 16k request, the TTFTs become (1.58s, 1.31s). With only a 0.05s increase in the 32k request's TTFT, the system's average/max TTFTs are reduced by 0.24/0.26s, respectively. However, an effective mechanism is still lacking to adaptively select the most suitable SP allocation based on the system's load conditions, under highly dynamic serving workloads.

Additionally, **static batching brings inefficient resource usage for decoding**. The resource utilization progressively declines as requests in a decoding batch complete execution. However, static batching precludes the addition of new requests during decoding, preventing the adoption of continuous batching to boost utilization [44, 46].

(3) SP Allocation Granularity: Request-level SP allocation cannot achieve both low TTFT and high resource utilization at the same time. **Allocating SP sizes by treating all tokens of a request as a whole**, as in LoongServe, provides an intuitive way to meet inter-request diverse resource demands. However, **in online serving with unpredictable request arrivals**, this strategy induces a trade-off between TTFT optimization and resource utilization: **Directly assigning large SP to long requests can cause resource idleness**, as SP's ring communication requires all instances to start computation simultaneously. When a long request arrives, a short request with a smaller SP size may already be running. To reduce TTFT, the scheduler may assign the long request a larger SP size by reusing instances occupied by the short request. In this case, the additional instances allocated to the long request remain idle during the short request's execution, hurting resource utilization. However, **allocating small SP for better resource utilization significantly degrades long requests' TTFT**, because larger SP sizes substantially reduce long requests' prefill latency.

For example, given 16 LLaMA3-8B SP instances (TP=1), if a 16k request is assigned SP=8 before the arrival of a 128k request, assigning SP=16 to the 128k request results in 8 instances idle for 0.31 seconds. However, directly assigning SP=8 using the 8 idle instances incurs a 1.34-second TTFT increase. This underscores the need for a fine-grained SP allocation strategy capable of jointly minimizing TTFT and maximizing resource utilization.

To address these limitations, we propose chunkwise dynamic sequence parallelism (CDSP) and build a distributed system, Tetris, to fully utilize CDSP for online long-context LLM serving. In the following sections, we will first present CDSP's basic concept and Tetris's system overview. Then, we will describe Tetris's inference engine and scheduler design. Finally, we will introduce Tetris's prototype implementation.

3 Tetris Overview

3.1 Chunkwise Dynamic Sequence Parallelism

As shown in Fig. 3-(a), request-level SP scheduling assigns SP uniformly to each request's all tokens. Although this approach tries to satisfy per-request resource demand, it creates imbalance across instances due to dynamic SP allocation. Such an imbalance results in instance idleness when allocating large SP sizes to reduce TTFT, as ring attention mandates simultaneous KV cache transfer across all instances. Conversely, decreasing SP size to mitigate resource idleness notably prolongs TTFT for long requests, whose prefill latency fluctuates by tens of seconds when shrinking SP sizes.

To fulfill requests' SP requirements without compromising resource utilization, we propose chunkwise dynamic sequence parallelism (CDSP), a more fine-grained

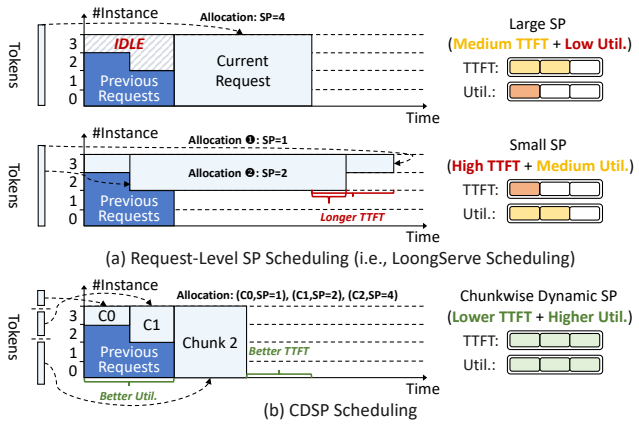

Figure 3. Basic concept of Chunkwise Dynamic SP (CDSP).

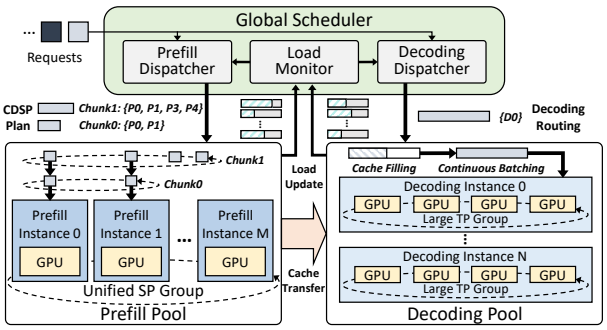

Figure 4. System Architecture of Tetris.

parallelism strategy. As depicted in Fig. 3-(b), rather than allocating a fixed SP size to the entire request, CDSP subdivides each request into multiple chunks and selects appropriate SP sizes for them. Specifically, CDSP applies larger SP to latter chunks to accommodate the computation demands of long requests. In contrast, preceding segments are scheduled with smaller SP sizes, allowing partial execution to start earlier by leveraging idle resource fragments. By progressively expanding the SP size across chunks — akin to filling the gaps in the tetris game — CDSP maximizes resource utilization and further reduces TTFT beyond request-level scheduling.

3.2 Serving System Overview

Design Goal: Tetris aims to enable fine-grained dynamic SP mechanism, while remaining fully compatible with SOTA optimization techniques. The cluster must satisfy distinct characteristics between prefill and decoding *(LoongServe Limitation (1))*. The scheduler must regulate SP allocation based on real-time system loads *(LoongServe Limitation (2))*, and the inference engine must fully optimize CDSP prefill computation *(LoongServe Limitation (3))*.

System Architecture: To this end, Tetris is built on prefill-decoding disaggregation, as shown in Fig. 4. In contrast to existing designs where all prefill instances operate independently, Tetris connects them into an identical SP group and assigns each a smaller TP size (e.g., TP=1), maximizing resource allocation flexibility. Each decoding instance adopts a larger TP size (e.g., TP=4 in Fig. 4) to fully optimize TBT. For each request, the prefill dispatcher generates CDSP execution plan based on real-time load conditions. The designated prefill instances conduct CDSP prefill and stream KV cache to the target decoding instance, which adds the request to continuous batching for output generation.

Although prefill-decoding disaggregation can alleviate *LoongServe Limitation (1)*, **existing designs are built solely on tensor/pipeline parallelism (TP/PP), lacking support for dynamic SP in disaggregation cluster** [31, 34, 46]. To fully utilize CDSP to solve *LoongServe Limitation (2),(3)*, Tetris must address the following challenges:

Challenge #1: Inference Engine Adaptation: *(1) Attention Computation.* As shown in Fig. 3-(b), SP size expansion results in uneven KV cache distribution, creating inter-instance load imbalance. Therefore, we need to tailor attention computation for CDSP to maximize its resource utilization. *(2) Cache Transfer Management.* Unlike existing non-SP disaggregated clusters, where each request's full KV cache is located on a single prefill instance, CDSP distributes each chunk's KV cache across multiple prefill instances. We need to coordinate cache transfer to ensure timely delivery of each request's all cache chunks to the decoding instance.

Challenge #2: Scheduler Customization: *(1) For CDSP Execution Plan*, we need to determine the chunk number, each chunk's token number, and the corresponding prefill instance allocation. They define a vast scheduling space given the large context window and numerous prefill instances. An efficient CDSP plan solver is vital to meet real-time requirements. *(2) For SP Size Regulation*, efficiently integrating real-time load impacts into the CDSP plan solver is also vital to achieve optimal global TTFT distribution.

The following sections will describe Tetris's solutions.

4 Tetris Inference Engine

4.1 CDSP Prefill Computaiton

Overall Procedure: As shown in Fig. 5, during CDSP computation, each chunk's tokens are evenly interleaved across the assigned prefill instance group. All instance groups compute sequentially following the chunk order. Before computing each chunk, the KV cache of all preceding chunks is evenly re-distributed to current chunk's instance group to balance the attention workload distribution. To reduce cache balancing overhead, we constrain that each chunk's instance group must include all instances involved in preceding chunks, which is ensured by the CDSP scheduler discussed later. In Fig. 5's two-chunk example, chunk-0 is first executed on instances *P0-P3*. Before chunk-1's execution, *P0-P3* forward the second half of their local KV cache to *P4-P7*, equalizing the cache load across chunk-1's instances.

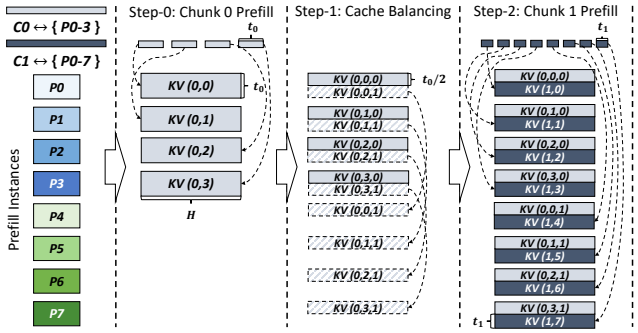

Figure 5. CDSP's Prefill Computation Procedure.

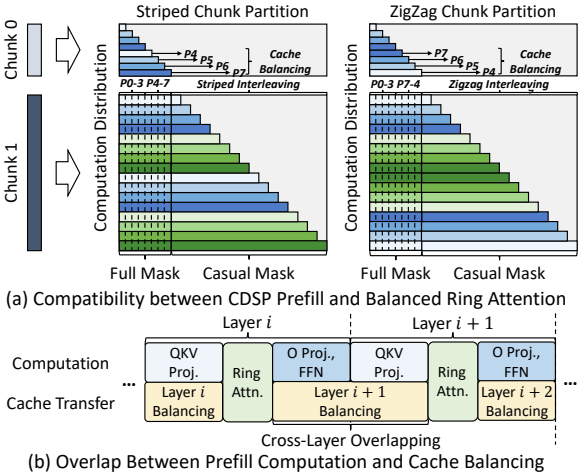

Figure 6. Optimizations for CDSP Prefill Computation.

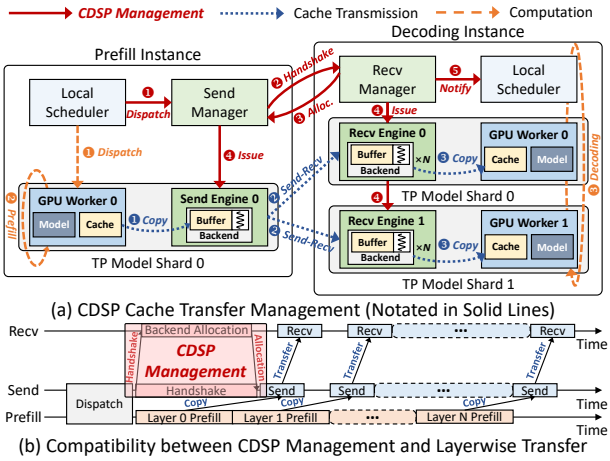

Figure 7. Handshake-based CDSP Transfer Management.

Cache-Balancing Simplification: Note that each chunk computes attention with all historical tokens. Therefore, as shown in Fig. 6-(a), balanced attention computation with preceding chunks only requires to split historical KV cache evenly on current instance group, regardless of each chunk's token interleaving strategy. Accordingly, we can still adopt striped/zigzag attention to achieve intra-chunk attention load balance, simplifying CDSP prefill's implementation.

Cache-Balancing Latency Overlap: Cache balancing introduces additional KV cache transfer. To eliminate its impact on TTFT, we propose a layer-wise overlap mechanism between prefill computation and cache balancing. The key insight is that fully connected layers perform computation independently of the KV cache. As shown in Fig. 6-(b), once the ring attention in current layer completes, its inter-instance communicator can be reused to perform cache balancing for the next layer. This cross-layer overlap efficiently hides cache balancing latency, ensuring to fully unveil CDSP's benefits.

4.2 CDSP Cache Transfer Management

Challenge: Backend Starvation. For each request, decoding instance begins computation only after receiving its full KV cache from all prefill instance groups. Since most transfer backends require GPU buffers [21, 26, 34], long-context serving, producing huge intermediate tensors, may leave insufficient memory to reserve a dedicated transfer backend for each prefill instance. Under this case, some instances may never obtain any backend without proper management, preventing the decoding instance from receiving the full KV cache. This starvation not only delays decoding execution, but also causes partially filled cache to occupy decoding instances for extended periods, reducing memory utilization.

Backend Allocation Handshake: To address this issue, we introduce a handshake mechanism into prefill-decoding cache transfer procedure. As shown in Fig. 7-(a), prefill instance's send manager initiates a handshake before issuing KV cache transfer (❷). If the receive engine is either buffer-free [6] or has sufficient backends, the handshake merely signals the receive manager to launch transfer using current prefill instance's dedicated backend. Otherwise, requests are sorted by the first handshake timestamp. The receive manager sequentially reserves backends for each request until all its chunks are transferred, preventing the starvation from interrupting latter chunks' transmission.

Overall Transfer Procedure: As shown in Fig. 7-(a), each request chunk is first dispatched to both the GPU workers (❶) and the send manager (❶). While GPU workers are computing (❷), the send manager issues a handshake to the target receive manager for backend allocation (❷). Once the allocation is confirmed (❸), both the send and receive managers issue cache transfer (❹). Then, send and receive engines use high-performance communication libraries [21, 26, 34] for transfer execution (❶-❸). After receiving all chunks' KV cache by repeating the above procedure, the receive manager will notify the local scheduler (❺) to insert the request into the decoding batch using iteration-level scheduling (❸).

Handshake Latency Overlap: As shown in Fig. 7-(b), since prefill computation is independent with handshake, the whole

Algorithm 1: CDSP Scheduling Algorithm

Input: unallocated prompt length L, previous chunk allocation A, SP size candidates S, prefill instance pool P.
Step 0: Initial (single-chunk) plan generation
$instance\_group \leftarrow SingleChunkSchedule(L, A, S, P)$
$opt\_allocation \leftarrow A.append((L, instance\_group))$
Step 1: Chunk plan exploration
$S_{CDSP} \leftarrow \{s_i | s_i \in S, s_i \leq |instance\_group|\}$
$SizePair \leftarrow \{(s_i, s_j) | s_i \in S_{CDSP}, s_j \in S_{CDSP}, s_i < s_j\}$
for *each* $(s_{current}, s_{next}) \in SizePair$ **do**
  // solve for current chunk's plan
  $current\_chunk\_plan \leftarrow GetChunkPlan(L, A, s_{current}, s_{next}, instance\_group)$
  if $Illegal(current\_chunk\_plan)$ **then**
    continue
  // generate full chunk plan recursively
  $L' \leftarrow L - current\_chunk\_plan.chunk\_length$
  $A' \leftarrow A.append(current\_chunk\_plan)$
  $S' \leftarrow \{s_i | s_i \in S_{CDSP}, s_i \geq s_{next}\}$
  $P' \leftarrow instance\_group.update(current\_chunk\_plan)$
  $chunk\_allocation \leftarrow CDSPSchedule(L', A', S', P')$
  // compare and update the best allocation record
  if $opt\_allocation.TTFT > chunk\_allocation.TTFT$ **then**
    $opt\_allocation \leftarrow chunk\_allocation$
return $opt\_allocation$

Algorithm 2: Single-chunk Scheduling Algorithm

Input: unallocated prompt length L, previous chunk allocation A, SP size candidates S, prefill instance pool P.
$(opt\_TTFT, opt\_group) \leftarrow (INF, \emptyset)$
// get previous chunks' token number and instance allocation
$C \leftarrow A.get\_total\_chunk\_length()$
$initial\_group \leftarrow A.get\_all\_instances()$
for *each* $s \in S$ **do**
  // extend previous allocation to generate new instance group
  $instance\_group \leftarrow GetGroup(P, initial\_group, s)$
  $T_{queue} \leftarrow \max_{T_i}\{T_i | p_i \in instance\_group\}$
  $T_{prefill} \leftarrow PerformanceModel(s, C, L)$
  $TTFT \leftarrow T_{queue} + T_{prefill}$
  // ensure sufficient performance gains to avoid over-expansion
  if $TTFT < opt\_TTFT \times (1 - improvement\_rate)$ **then**
    $(opt\_TTFT, opt\_group) \leftarrow (TTFT, instance\_group)$
return opt_group

handshake procedure (❷-❸ in Fig. 7-(a)) can be seamlessly integrated into layer-wise cache transmission [31, 34]. In this way, we can overlap the handshake with prefill computation to efficiently hide its latency overhead.

5 Tetris Scheduling Algorithm

5.1 CDSP Prefill Scheduling

Prefill Latency Model: Given LLMs' huge context windows, exhaustive chunk size searching leads to prohibitive scheduling complexity. Therefore, we follow previous works' practice [42, 46] and adopt a latency model based on floating point operations (FLOPs) to guide scheduling. For a request chunk R, denote its historical token number as C, and the token number within it as L. The prefill latency under the SP size of s can be estimated as:

$$T_s(R) = a_s + b_s \cdot L + c_s \cdot (C \cdot L) + d_s \cdot L^2, \tag{1}$$

where a_s, b_s, c_s, d_s are coefficients for the overhead of constant factors, fully-connected layers, attention with historical tokens, and attention within current tokens, respectively. The latency model of each target SP size can be obtained from least-squares fitting by collecting latency data across various (C, L) pairs. This fitting process can be performed offline, and the performance models can be reused during subsequent online serving until the GPU/model type changes.

Overall Scheduling Workflow: As summarized in Algorithm 1, CDSP's scheduling employs a recursive approach to search for the optimal chunking strategy. It takes four inputs: (1) Unallocated token number L. (2) Previous chunk allocation $A = [a_0, ..., a_{l-1}]$, where a_i records chunk i's token number and prefill instance group. For a new request (i.e., the first invocation of Algorithm 1), A is initialized as an empty list. (3) The candidate set of SP sizes $S = \{s_0, ..., s_{m-1}\}$, where each s_j denotes an available SP size for allocation. (4) The prefill instance pool $P = \{p_0, ..., p_{n-1}\}$, where each p_k maintains the queuing time T_k when the remaining tokens are scheduled for execution.

Given these inputs, the algorithm first treats all remaining tokens as a single chunk to conduct initial instance group allocation (details will be discussed later), which determines the max SP size according to real-time request pressure (**line 3-4**). Then, the algorithm further investigates the gains from CDSP chunking. It enumerates all valid SP size pairs for the current and subsequent chunks, according to the instance number of the initial allocation (**line 6-7**). For each pair, the algorithm first solves current chunk's execution plan based on $s_{current}$'s corresponding instance subgroup (details will be discussed later) (**line 10**). It then filters out illegal plans, such as those with negative chunk sizes or chunk lengths that are too short to yield benefits under $s_{current}$ (**line 11-12**). If *current_chunk_plan* is valid, the algorithm modifies input states and recursively solves for the complete chunk allocation (**line 14-18**). To avoid double-counting historical queuing delays, *instance_group*'s queuing latency must be updated before each recursive call. Assume *current_chunk_plan*'s prefill computation latency and max instance queuing latency are $T_{prefill}$ and T_{queue}, respectively. For each instance $p_i \in instance\_group$, its queuing latency T_i is updated as follows:

$$T_i \leftarrow max\{0, T_i - (T_{queue} + T_{prefill})\} \tag{2}$$

When S_{CDSP} contains only one candidate, the recursive search terminates and directly returns the single-chunk plan. After recursive searching returns, the algorithm updates the best allocation record based on the TTFT estimation (**line 20-21**). Once all SP pairs in *SizePair* are explored, the algorithm returns the optimal allocation (**line 22**).

Single-chunk Scheduling (line 3 in Algorithm 1): As listed in Algorithm 2, for each SP size s, it constructs instance group by extending the instance set allocated to previous chunks (**line 8**), reducing cache balancing overhead as discussed in Sec. 4.1. It then estimates the TTFT by combining the prefill latency predicted by Eq. (1) with the max instance queueing latency (**line 9-11**), which is used to update the best allocation (**line 13-14**). Specifically, to avoid excessive SP expansion, the algorithm increases SP size only when the TTFT gain exceeds a certain threshold, which is dynamically adjusted based on real-time request arrival pressure.

The **instance group extension** (i.e., *GetGroup* in line 8) proceeds as follows: *(1) When initial_group is empty* (i.e., first-chunk allocation), the algorithm first checks whether s can be satisfied within a single node. If so, it selects the node with the minimal s-th shortest queuing latency and takes its s shortest-queued instances to avoid cross-node fragmentation. Otherwise, if s spans k full nodes, the algorithm selects the top-k nodes with the shortest queuing latency. For remaining instances, the same intra-node selection strategy is applied across the unallocated nodes. *(2) When initial_group is non-empty*, the algorithm first adds instances within the nodes containing *initial_group*'s instances. If additional instances are still needed, the algorithm applies the same strategy as (1) to the remaining free nodes.

To select **real-time load-aware improvement rate**, we implement a simulator-based search mechanism. The key insight is that the request length distribution of long-context services remains stable over days or weeks. Therefore, we can periodically collect the length distribution and sample requests under various request arrival rates to simulate different load conditions. For each arrival rate, we can use Eq. (1) to simulate TTFT under various improvement rates, yielding the one that minimizes TTFT. This profiling can be performed offline. During online serving, the scheduler monitors the request rate within a sliding time window and dynamically updates the improvement rate by querying the pre-profiled optimal rate records.

Chunk Plan Solving (line 10 in Algorithm 1): As listed in Algorithm 3, it first allocates instance groups to $s_{current}$ and s_{next} using the extension strategy discussed above (**line 6-7**). Then, the algorithm sets the current chunk's prefill latency budget as the difference between the queuing delays of *next_group* and *current_group* (**line 9-11**). For example, in the case shown in Fig. 3-(b), when solving the plan for chunk 1 with $s_{current}$=2 and s_{next}=4, the budget is obtained by comparing the maximum queuing latencies of instances 0–3 and 2–3. Given the latency budget and the historical token number, the performance model in Eq. (1) becomes a polynomial in the chunk size, which can be solved numerically (e.g., using Newton's method) to determine the current chunk's token number (**line 13-14**).

Algorithm 3: Chunk Plan Solving Algorithm

1 **Input:** unallocated prompt length L, previous chunk allocation A, current chunk's SP size $s_{current}$, subsequent chunks' minimal SP size s_{next}, prefill instance pool P.
2 // get previous chunks' token number and instance allocation
3 $C \leftarrow A.get\_total\_chunk\_length()$
4 $initial\_group \leftarrow A.get\_all\_instances()$
5 // get current and next instance groups
6 $current\_group \leftarrow GetGroup(P, initial\_group, s_{current})$
7 $next\_group \leftarrow GetGroup(P, current\_group, s_{next})$
8 // estimate chunk computation latency budget
9 $T_{queue}^{current} \leftarrow \max_{T_i}\{T_i | p_i \in current\_group\}$
10 $T_{queue}^{next} \leftarrow \max_{T_j}\{T_j | p_j \in next\_group\}$
11 $T_{budget} \leftarrow T_{queue}^{next} - T_{queue}^{current}$
12 // use performance model to solve chunk size
13 $L_{chunk} \leftarrow min(L, SolvePerformanceModel(T_{budget}, s_c, C))$
14 **return** $(L_{chunk}, current\_group)$

5.2 Decoding Scheduling

Since decoding instances operate independently, we can reuse existing scheduling strategies [34, 36, 46]. Currently, we extend the "virtual usage" proposed by Llumnix [36] in decoding scheduler: The KV cache slots of requests with ongoing cache transfer is treated as virtual sage. During scheduling, each new request is routed to the instance with the highest freeness rate, defined as the ratio between available slots (excluding virtual usage) and the active batch size. To improve load estimation accuracy, the scheduler updates slot statistics each time a request returns its decoding output.

6 Implementation

Tetris's serving framework is implemented with ~17.5K lines of code based on C++ and Python, including an API frontend, a control plane, and an inference backend. The frontend adopts FastAPI [9] to receive requests, and provides an interface to update improvement rate when request distribution shifts. The control plane contains a global manager and each instance's local managers. The global manager is mainly implemented with Python, with the CDSP scheduler (Algorithm 1) written in C++ to eliminate scheduling latency. Ray [23] is used to communicate between the global manager and model instances. Each instance's local managers are assigned to distinct Python coroutines, which use Ray to manage computation or KV cache transmission.

The inference backend is build on Pytorch [30] and Triton-distributed [45], and reuses some components of vLLM [18]. For prefill computation, we extend Flash Attention [7] to support zigzag ring attention for historical tokens, and use NVSHMEM [25] to reduce ring communication overhead. For decoding computation, we adopt Flash Decoding [8] for attention and use CUDAGraph [33] to eliminate kernel launch overhead. CDSP cache balancing and prefill-decoding cache transfer are implemented with NCCL [26], which has supported concurrent communicator execution since

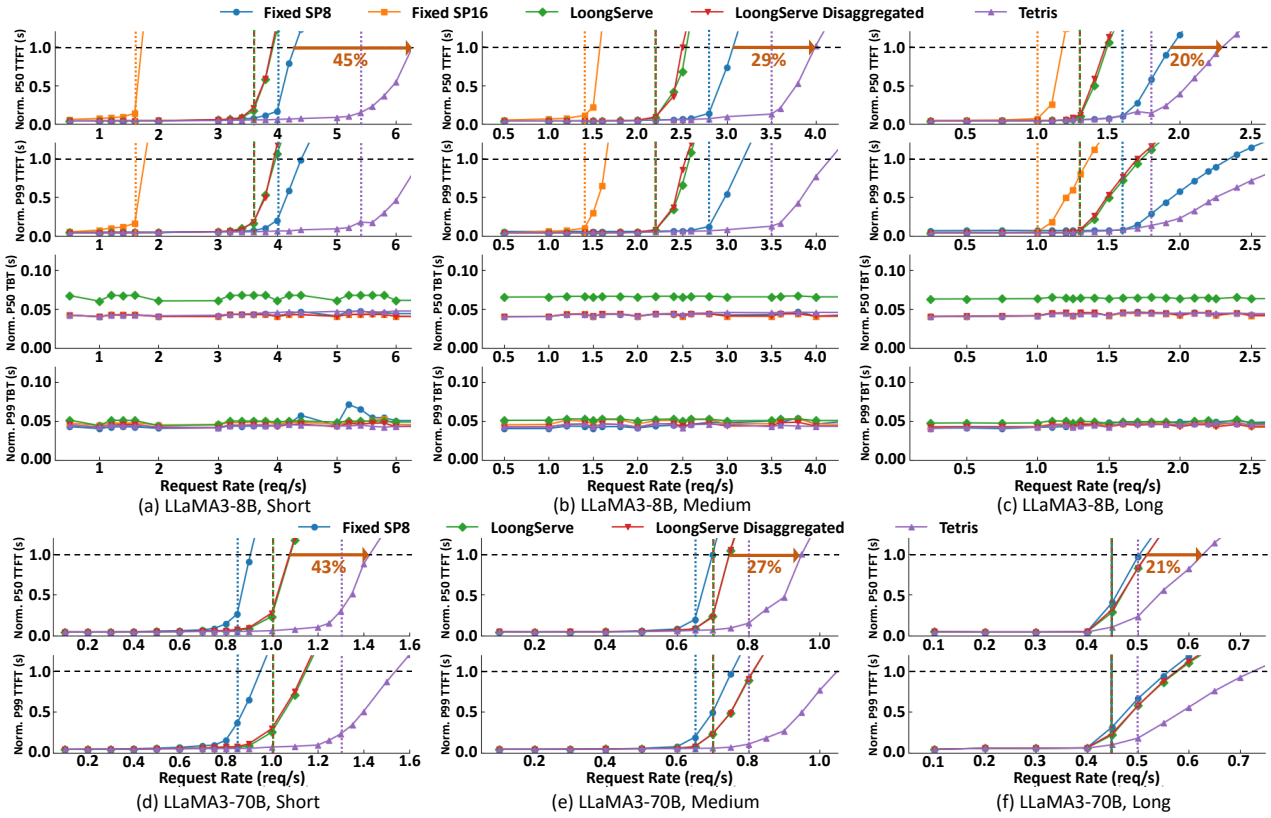

Figure 8. Comparison against Baselines on LLaMA3-8B/70B under Different Workloads.

v2.26 [27]. We reserve dedicated buffers and CUDA streams for cache transfer to improve bandwidth utilization.

Tetris also contains a simulator-based improvement rate profiler implemented with ~2.1K lines of Python. For each request rate, the simulator generates timestamps using a Poisson process and samples requests from the given length distribution. It then simulates prefill execution as discrete events [35] using latency models. After comparing TTFTs under different improvement rates, the simulator identifies the optimal improvement rates for the CDSP scheduler.

7 Evaluation

7.1 Experiment Setup

Model: To evaluate Tetris's performance at different scales, we use LLaMA3-8B and LLaMA3-70B [14] models. We employ their context-extended variants with RoPE scaling [37] to support the context window in our workloads.

Testbed: We conduct experiments on A100 GPU clusters. Each node contains eight NVIDIA-A100-SXM4-80GB GPUs connected with NVLINK, 128 CPU cores, 2TB host memory, and eight 200 Gbps InfiniBand NICs. We deploy LLaMA3-8B on four nodes and LLaMA3-70B on eight nodes.

Workload: We collect three real-world request traces with different length distributions from our production service. Specifically, the **Short** trace's sequence length ranges from 4k to 95k, with an average length of 23.6k. The **Medium** trace's sequence length ranges from 8k to 142k, with an average length of 32.8k. The **Long** trace's sequence length ranges from 16k to 190k, with an average length of 50.1k.

Metric: As discussed in Sec. 2.2, we adopt TTFT and TBT, the key metrics for online LLM serving, to measure each system's performance. We report both P50 and P99 values to characterize the overall latency distribution.

Baseline: We compare Tetris with the following baselines:

(1) LoongServe [42]: It is the first and the only SP-enabled long-context LLM serving framework. Moreover, it reports state-of-the-art long-context LLM serving performance compared with existing best-performing non-SP serving systems [1, 18, 22, 46]. We set TP=1 for LLaMA3-8B and TP=4 for LLaMA3-70B to maximize its flexibility (i.e., ESP size) while ensuring sufficient cache slots on each instance. To avoid TTFT interference as discussed in Sec. 2.4 (*Limitation (2)*), we adopt single-request scheduling to minimize its TTFT.

(2) LoongServe Disaggregated: This is a prefill-decoding decoupled cluster similar to Tetris's architecture, while the prefill scheduler adopts LoongServe's single-request scheduling. We set the P/D ratio to 1:1 after carefully balancing TTFT and TBT. For LLaMA3-8B, the TP sizes of prefill and decoding instances are 1 (identical to LoongServe) and 8. For LLaMA3-70B, since decoding latency reports marginal

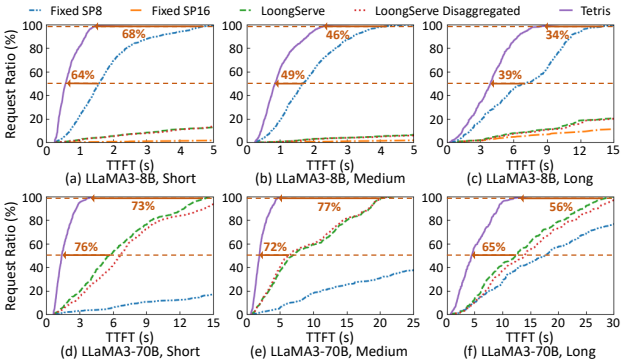

Figure 9. TTFT Distribution Analysis.

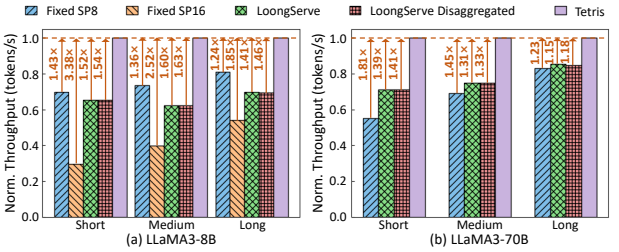

Figure 10. Throughput Analysis under TTFT Constraints.

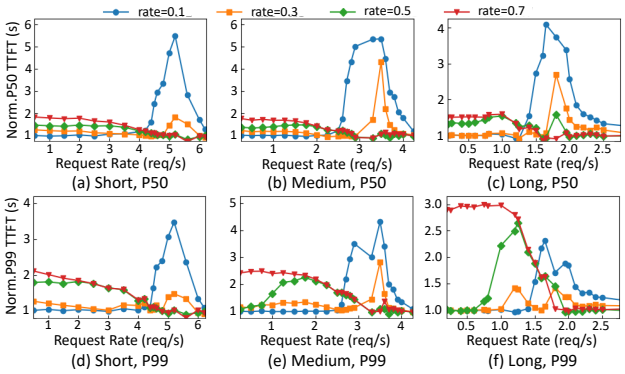

Figure 11. Improvement Rate Analysis on LLaMA3-8B.

improvement beyond TP=4, we set TP size to 4 (identical to LoongServe) for all instances and focus on TTFT evaluation. **(3) Fixed-SP Scheduling**: It also adopts the prefill-decoding disaggregation architecture, where prefill instances are organized into multiple independent SP groups. We evaluate fixed SP sizes of 8 and 16, co-locating each group's instances on the same node where possible. Requests are scheduled to the group with the lowest queuing delay, which is estimated using Eq. (1). The P/D ratio and TP size allocation are identical to LoongServe Disaggregated.

For Tetris, we also adopt the same P/D ratio and TP size allocation as LoongServe Disaggregated for fair comparison. The SP size candidates are set to powers of two to reduce resource fragmentation. We adopt the simulator to collect optimal improvement rates (ranging from 0.05 to 0.75) for request rates incremented by 0.5 req/s. During serving, the improvement rate is updated every 30 seconds. The scheduler selects the recorded request rate closest to the observed value and applies the corresponding optimal improvement rate.

7.2 Comparison against Baselines

We first compare Tetris with the baselines through stress tests on the collected real workloads, where different load conditions are simulated by scaling the request arrival timestamps. Similar to LoongServe [42], we normalize all results to 25× of the light-load latency. As shown in Fig. 8, for LLaMA3-8B, fixing the SP size to 16 reports the worst TTFT due to the resource over-provision. It not only degrades short requests' TTFTs but also postpones subsequent requests' execution. Shrinking the fixed SP size to 8 improves TTFT. However, it hurts long requests' TTFTs and remains inflexible for short requests, as SP-8 can still over-allocate resources for their demands. LoongServe and LoongServe Disaggregated perform between the two fixed-SP configs. Although they can mitigate TTFT degradation for short requests, excessive SP expansion still delays request execution and hurts overall TTFT. Besides, although LoongServe exposes all instances to the prefill scheduler via ESP, it must reserve dedicated instances for decoding batches, resulting in marginal performance gains over LoongServe Disaggregated. Compared with the best-performing baseline (i.e., Fixed SP 8), Tetris can increase the max load by 20%-45%, owing to its fine-grained SP adjustment and prudent control of SP expansion. As to TBT, although LoongServe reports comparable P99 latency, its P50 latency is 55%-67% higher than the large-TP configuration enabled by the disaggregated architecture.

For LLaMA3-70B, since prefill adopts TP-4 and decoding reports marginal TBT gains from TP-4 to TP-8, we mainly compare the TTFT results. LoongServe (Disaggregated) can outperform Fixed SP8, as SP-8 is already an over-provision for short requests under TP-4. Compared with these baselines, Tetris enhances the max load by 21%-43%. CDSP remains effective as model and system scales increase.

7.3 Performance Analysis and Ablation Study

TTFT Distribution Analysis: To analyze Tetris's TTFT benefits, we compare the cumulative TTFT distributions under the highest request rate where the best-performing baseline maintains low latency to preserve user experience. Each system's critical request rates are marked by vertical dashed lines in Fig. 8. As Fig. 9 shows, Tetris achieves 1.64-2.78×/2.86-4.17× lower P50 TTFT on LLaMA3-8B/70B. As to P99 TTFT, it yields 1.52-3.13×/2.27-4.35× lower values, respectively. Tetris can effectively enhance the serving quality compared with existing SOTA systems.

Throughput Analysis: To assess Tetris's resource efficiency, we then compare all systems' throughput under their critical request rates. As shown in Fig. 10, Tetris improves the

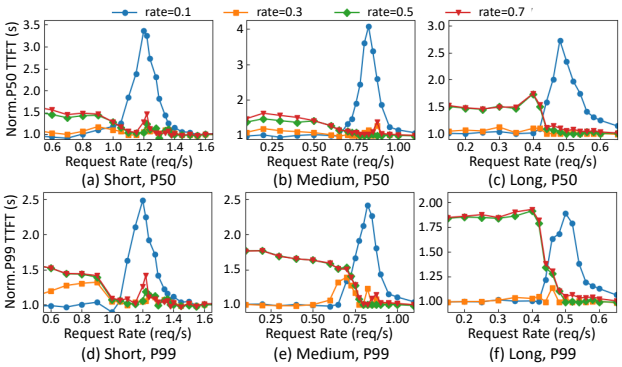

Figure 12. Improvement Rate Analysis on LLaMA3-70B.

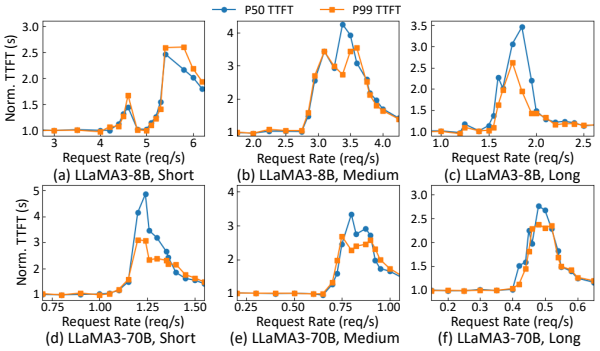

Figure 13. TTFT Slowdown under Single-Chunk Scheduling.

throughput by 1.24-3.38×/1.15-1.81× for LLaMA3-8B/70B, while maintaining low latency for user experience. The fine-grained and moderate SP allocation in Tetris can better adapt to varying request lengths, enhancing resource utilization.

Improvement Rate Analysis: To analyze how improvement rate preferences vary with loads, we compare Tetris's TTFT under different fixed rates, which span the range used in rate exploration. All results are normalized to the TTFT under dynamic rate adjustment.

As shown in Fig. 11-12, under low request rates, TTFT is dominated by prefill latency. Therefore, enforcing a smaller improvement rate (e.g., 0.1, 0.3) helps allocate larger SP sizes, reducing computation time and improving overall TTFT. As request load increases, queuing delay becomes a larger contributor to TTFT. Increasing the improvement rate (e.g., 0.5, 0.7) mitigates excessive SP expansion, enabling earlier execution of later requests and reducing queuing-driven TTFT. When the system is highly saturated, queuing delay constitutes the majority of TTFT, rendering it less sensitive to rate variation. Compared with fixed-rate settings, our dynamic rate adjustment can select near-optimal rates across varying load conditions, enabling CDSP to effectively optimize TTFT.

Chunking Analysis: To quantify the benefits of CDSP chunking, we compare CDSP scheduling with single-chunk scheduling (i.e., skipping line 5-21 in Algorithm 1). As shown

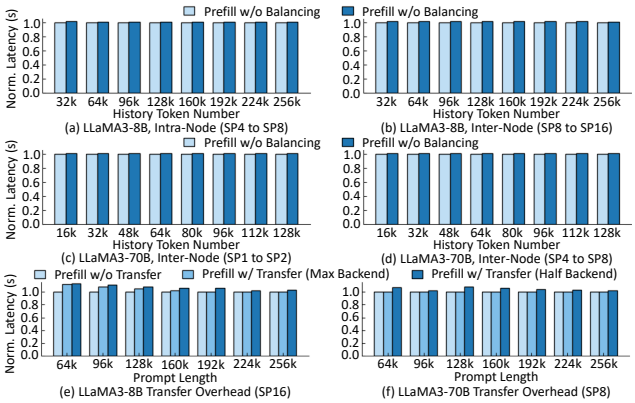

Figure 14. Cache Transfer Overhead Analysis.

Table 2. Scheduler Overhead under Different SP Sizes.

Max SP Size	8	16	32	64	128
Avg./Max Latency (us)	22.8/52.5	25.8/86.8	22.9/53.4	24.9/45.1	30.6/73.7

in Fig. 13, single-chunk scheduling incurs up to 2.33-4.17×/2.71-4.77× higher P50 TTFT on LLaMA3-8B/70B. For P99 TTFT, it yields 2.64-3.58×/2.43-3.23× higher values, respectively. Under light loads, each request's minimal queuing delay limits CDSP's search space and makes single-chunk plan efficient enough. As the load increases, queuing latency becomes more pronounced, and the resource fragmentation intensifies. Therefore, CDSP's fine-grained SP allocation can significantly improve resource efficiency and reduce TTFT. When the system is highly saturated, similar to the improvement rate, accumulated queuing delays reduce the system's sensitivity to chunking, leading to diminishing TTFT gains.

7.4 Overhead Analysis

CDSP Cache Balancing: To evaluate the overhead under different length ratios, we set current chunk's token number to 128k/64k for LLaMA3-8B/70B, and vary the historical token number from 25% to 2× of it. For each setting, we test both intra-node and inter-node overheads. As shown in Fig. 14-(a)~(d), CDSP balancing only incurs up to 1.8% extra overhead, proving the efficiency of the overlap strategy.

CDSP Handshake: To assess the multi-instance cache transfer overhead, we first test under the largest SP sizes with max backend allocation. Since the capacity is sufficient under our settings, each prefill instance can be assigned a dedicated backend. As shown in Fig. 14-(e)~(f), cache transfer incurs 0.6%-11.8% (average 2.1%) overhead. We then halve the backend number to conduct stress tests under limited capacity, which results in only 1.5%-5.4% (average 3.8%) additional RPC overhead. The handshake-based management mechanism can efficiently utilize buffer-backed transfer backends.

CDSP Scheduling: To evaluate the efficiency of CDSP prefill scheduling, we measure its execution latency under different

SP sizes by randomly sampling request length and instance queuing latency. Each SP size is tested 1000 times. As listed in Table 2, even when SP=128, the scheduling latency remains ≤86.8us, proving Algorithm 1's efficiency in meeting the real-time requirements of online serving.

8 Conclusion

This paper proposes Tetris, a serving system empowered by chunkwise dynamic sequence parallelism (CDSP) for online long-context LLM serving. CDSP's fine-grained SP allocation satisfies diverse resource demands while maximizing resource utilization. With the load-aware scheduling, Tetris fully unveils CDSP's benefits under dynamic online workloads. Experiments on real-world workloads shows that Tetris achieves up to 4.35× lower TTFT than existing SOTA systems and increases max serving capacity by up to 45%.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[2] DeepSeek AI. 2025. DeepSeek. https://chat.deepseek.com/.

[3] Anthropic. 2025. All models overview. https://docs.anthropic.com/en/docs/about-claude/models/all-models.

[4] Anthropic. 2025. Claude. https://www.anthropic.com/claude.

[5] 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).

[6] Shiyang Chen, Rain Jiang, Dezhi Yu, Jinlai Xu, Mengyuan Chao, Fanlong Meng, Chenyu Jiang, Wei Xu, and Hang Liu. 2024. KVDirect: Distributed Disaggregated LLM Inference. *arXiv preprint arXiv:2501.14743* (2024).

[7] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[8] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash-decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

[9] FastAPI. 2025. FastAPI. https://github.com/tiangolo/fastapi.

[10] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. 2025. ByteScale: Efficient Scaling of LLM Training with a 2048K Context Length on More Than 12,000 GPUs. *arXiv preprint arXiv:2502.21231* (2025).

[11] Github. 2025. Copilot. https://github.com/features/copilot.

[12] Google. 2025. Gemini. https://gemini.google.com/app.

[13] Google. 2025. Gemini 2.5 Pro. https://deepmind.google/technologies/gemini/pro/.

[14] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh,

Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[15] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, Yonggang Wen, Tianwei Zhang, Xin Jin, and Xuanzhe Liu. 2024. Loongtrain: Efficient training of long-sequence llms with head-context parallelism. *arXiv preprint arXiv:2406.18485* (2024).

[16] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509* (2023).

[17] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[18] 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 *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[19] Dacheng Li, Rulin Shao, Anze Xie, Eric Xing, Joseph E Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. Lightseq: Sequence level parallelism for distributed training of long context transformers. (2023).

[20] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring attention with blockwise transformers for near-infinite context, 2023. *URL https://arxiv. org/abs/2310.01889* (2023).

[21] Meta. 2025. Gloo collective communication library. https://github.com/facebookincubator/gloo.

[22] Microsoft. 2025. DeepSpeed Model Implementations for Inference (MII). https://github.com/deepspeedai/DeepSpeed-MII.

[23] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[25] NVIDIA. 2025. NVSHMEM. https://docs.nvidia.com/nvshmem/api/using.html.

[26] NVIDIA. 2025. Optimized primitives for collective multi-GPU communicatio Resources. https://github.com/NVIDIA/nccl.

[27] NVIDIA. 2025. Using multiple NCCL communicators concurrently. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html#using-multiple-nccl-communicators-concurrently.

[28] OpenAI. 2025. Chatgpt. https://chatgpt.com/.

[29] OpenAI. 2025. GPT-4o. https://platform.openai.com/docs/models/gpt-4o.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*.

[31] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[32] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. arXiv:2211.05102 [cs.LG] https://arxiv.org/abs/2211.05102

[33] PyTorch. 2025. CUDAGraph. https://pytorch.org/docs/stable/generated/torch.cuda.CUDAGraph.html.

[34] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation—A {KVCache-centric} Architecture for Serving {LLM} Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 155–170.

[35] Stewart Robinson. 2014. *Simulation: the practice of model development and use*. Bloomsbury Publishing.

[36] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 173–191.

[37] Gradient Team. 2024. Scaling Rotational Embeddings for Long-Context Language Models. https://www.gradient.ai/blog/scaling-rotational-embeddings-for-long-context-language-models.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[39] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jiashi Li, Faming Wu, and Bin Cui. 2025. FlexSP: Accelerating Large Language Model Training via Flexible Sequence Parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 421–436.

[40] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[42] 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 *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 640–654.

[43] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. 2024. Context parallelism for scalable million-token inference, 2024. *URL https://arxiv. org/abs/2411.01783* (2024).

[44] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[45] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. 2025. Triton-distributed: Programming Overlapping Kernels on Distributed AI Systems with the Triton Compiler. *arXiv preprint arXiv:2504.19442* (2025).

[46] 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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.