


EFFICIENT LONG-CONTEXT LANGUAGE MODEL TRAINING BY *Core Attention Disaggregation*

Yonghao Zhuang [*1] **Junda Chen** [*2] **Bo Pang** [†2] **Yi Gu** [2 3] **Yibo Zhu** [4] **Yimin Jiang** [4]
Ion Stoica [5] **Eric Xing** [1 3] **Hao Zhang** [2]

ABSTRACT

We present core attention disaggregation (CAD), a technique that improves long-context LLM training by disaggregating the core attention (CA) – the parameter-free $\mathrm{softmax}(\mathbf{QK}^\top)\mathbf{V}$ computation – and schedules it on an independent pool of resources. Existing systems co-locate core attention with other components. At long context, the quadratic growth of CA computation and near-linear growth of the rest create load imbalance – hence stragglers across data and pipeline groups. CAD is enabled by two key observations: (i) *statelessness*: CA has no trainable parameters and minimal transient state, so balancing reduces to scheduling compute-bound tasks; and (ii) *composability*: modern attention kernels sustain high utilization on fused batches of arbitrary-length token-level shards. CAD dynamically partitions the core attention computation into token-level tasks (CA-tasks), and dispatches them to a pool of devices specialized for CA computation (attention servers). It then rebatches CA-tasks to equalize CA compute across attention servers without loss of kernel efficiency. We have implemented CAD in a system called DistCA with a ping-pong scheme to completely overlap communication with compute, and in-place attention servers to improve memory utilization. Scaling to 512 H200 GPUs and 512K context length, DistCA eliminates DP/PP stragglers, achieves near-perfect compute and memory balance, and improves end-to-end training throughput by up to 1.9× over Megatron-LM and 1.35× over existing load-balancing methods

1 INTRODUCTION

Recent large language model (LLM) applications show a steadily increasing demand for processing longer contexts. For instance, reasoning workloads must generate long chain-of-thoughts to yield accurate answers (Guo et al., 2025); coding agents operate over multi-file repositories (Liu et al., 2024b). To reliably support these use cases, LLMs must operate with contexts of 100K - 1M tokens at inference.

Equipping an LLM with long-context capability typically requires training on datasets with both long documents and ordinary (short) documents (Gao et al., 2025). A standard approach to batch documents of variable length is *document packing* (Rae et al., 2021; Wang et al., 2024): concatenate multiple documents into a fixed-size chunk and apply an attention mask to block cross-document attention. While document packing improves throughput, it leads to variance in the attention compute – therefore *load imbalance* – across different chunks (Lin et al., 2025). The root cause is that self-attention compute in transformers grows quadratically with sequence length, whereas the rest scales approximately linearly. Consequently, two chunks with the same total tokens can incur very different attention FLOPs depending on how documents are distributed. For example, in Figure 1, a chunk with a single 4K-token document requires roughly 4x the attention FLOPs of a chunk packed with four 1K-token documents, even though both contain 4K tokens.

This imbalance manifests as stragglers (Figure 1) in large-scale distributed training in two ways (Wang et al., 2025c; Lin et al., 2025). First, in data parallelism (DP), replicas process different chunks and synchronize at the gradient barrier; the replica with the largest attention workload stalls the others. Second, in pipeline parallelism (PP) (Huang et al., 2019; Shoeybi et al., 2019), pipeline stages operate on different microbatches concurrently; a microbatch with a larger attention workload makes its current stage a straggler, creating pipeline bubbles which stall the entire pipeline. In hybrid DP and PP, these effects compound, and slowdowns of 1.34-1.44x (Wang et al., 2025c; Lin et al., 2025) have been reported even under modest context lengths.

One remedy is to equalize compute by assigning more tokens to devices processing shorter documents (Wang et al., 2025c). This balances compute but unbalances memory, as activation footprints grow with total tokens. In the "4×1K

[*]Equal contribution † Work done at UCSD. [1]Carnegie Mellon University [2]UC SanDiego [3]MBZUAI [4]StepFun [5]UC Berkeley. Correspondence to: Eric Xing <epxing@andrew.cmu.edu>, Hao Zhang <haz094@ucsd.edu>.

Proceedings of the 9^{th} MLSys Conference, Bellevue, WA, USA, 2026.

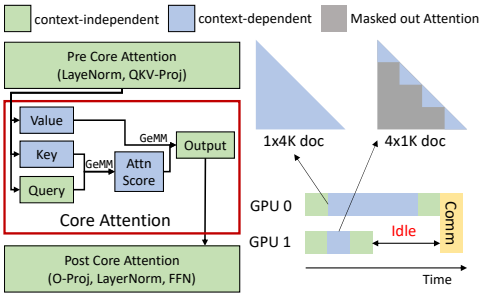


Figure 1. Transformer and its workload imbalance caused by core attention.

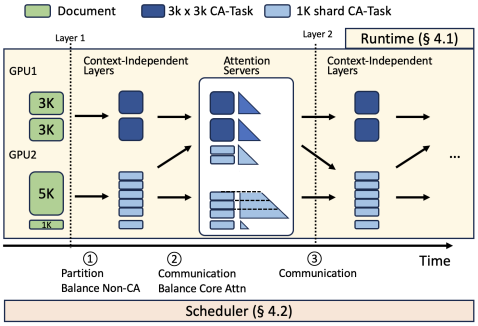


Figure 2. DistCA architecture.

Table 1. Compute and Memory for components in LLMs. l is the document's number of tokens.

	Memory	Compute
CA	0	$O(l^2)$
Linear	$O(l)$	$O(l)$
MISC	$O(l)$	≈ 0

CA: core attention layer;
Linear: FFN, qkvo-proj
MISC: layer norm, dropout, ...

vs. 1×4K" example: matching the 1×4K attention FLOPs requires 12 more 1K documents on the 4×1K device, raising its activation memory by 3×. Another remedy, context parallelism (CP) (Liu et al., 2024a; Jacobs et al., 2023), shards each document along the sequence dimension and equally distributes shards across DP replicas, which balances compute and memory, but at the cost of extra communication of KV states. Unfortunately, it cannot mitigate PP stragglers: pipeline stages hold disjoint model layers and cannot jointly compute shards of the same document; straggler microbatches still generate bubbles that stall other stages.

Fundamentally, the imbalance stems from mismatched complexity between attention and the rest of the model, as illustrated in Table 1. When attention and other layers are collocated, the mismatch grows with model scale and context length, leading to a severe load imbalance.

This naturally leads us to disaggregate attention from the rest of the model, so the attention and non-attention components can be scaled *independently* to balance workload across devices. A challenge, however, is that disaggregation introduces *per-layer* transfers of inputs and outputs across the attention boundary. At first glance, this communication is prohibitive and could negate its benefits. Surprisingly, we find the opposite: precisely isolating the *core attention* (CA) – the weightless $\mathrm{softmax}(\mathbf{QK}^\top)\mathbf{V}$ computation (Figure 1) – and applying targeted overlap and placement optimizations allows the communication to be effectively hidden in today's long-context training workloads.

Specifically, we identify two key characteristics of core attention that makes disaggregation effective. First, *statelessness*: CA contains no trainable parameters and stores only small per-row softmax statistics; balancing CA hence reduces to scheduling compute-bound tasks. Second, *composability*: CA can be partitioned at token granularity into arbitrary-length shards. Each shard, given its target tokens' $\mathbf{Q}$ and context tokens' $\mathbf{K}$, $\mathbf{V}$, independently computes the output. Shards from different documents can be re-batched into a single high-occupancy kernel. In modern attention kernels (e.g., Flash Attention), throughput primarily depends on the aggregated tokens in the fused call rather than their document of origin. This allows partitioning and recombining shards to equalize CA compute, instead of committing to the uniform splits typically assumed by CP.

These observations lead to *core attention disaggregation* (CAD), which disaggregates CA and schedules it independently on a specialized pool of resources, named *attention servers*. Attention servers are stateless, and accept core attention tasks – the CA computation of *arbitrarily partitioned* document shards – as compute requests. The runtime dynamically batches CA tasks into large fused kernel calls and schedule them to any attention server. Hence, CAD allows to achieve *near-perfect* load balancing across all attention servers, while avoiding memory imbalance. Unlike CP, CAD decouples core attention from the parallelisms for the rest of the model, eliminating stragglers in DP and PP.

Implementing CAD raises two practical challenges. First, CA is compute-intensive but memory-light; dedicating devices as attention servers might underutilize memory. We therefore time-share GPUs between attention servers (Li et al., 2014) and the rest of model computation to keep both compute and memory utilization high (§4.1). Second, dispatching token-level core attention tasks adds communication overhead; we design a ping-pong execution scheme (§4.1) to overlap communication with computation between two alternating batches. We also develop a scheduler (§4.2) that optimizes document sharding to strike a balance between balancing workloads while minimize communication.

We implement CAD in a system called DistCA, and evaluate it at up to 512 H200 GPUs for workloads with up to 512k context length. DistCA improves end-to-end throughput by up to 1.9x against Megatron-LM, and 1.35x against existing load balance methods. Our studies show that the communication caused by CAD can be fully hidden while attention is near-perfectly balanced.

In summary, this paper makes three key contributions:

- Propose core attention disaggregation to address the load imbalance in long-context training of LLMs.
- Implement DistCA with three key optimizations: in-place GPU time sharing, ping-pong overlap, and a workload-

balanced and communication-aware scheduler.
- Conduct comprehensive evaluation of CAD on large-scale realistic training workloads.

2 BACKGROUND

2.1 LLM Architecture

LLMs adopt the Transformer architecture, which consists of a stack of identical Transformer layers. As shown in Figure 1, we distinguish two types of layers.

Context-independent layers include QKV-projection, output-projection, feed-forward network (FFN), layernorm, and position embedding. These layers operate token-wise: the output for each token depends only on its own hidden state, so both compute and activation memory scale approximately linearly with the number of tokens. The dominant cost arises from the linear (GEMM) operators.

Context-dependent layers include only core attention (CA). In this paper, we deliberately distinguish *core attention* from *attention*; the latter refers to the composite of QKV-projection, output-projection, layernorm, and core attention in most literature. Given queries ($\mathbf{Q}$), keys ($\mathbf{K}$), and values ($\mathbf{V}$), CA computes attention scores $\mathbf{P} = \mathrm{softmax}(\mathbf{Q}\mathbf{K}^\top)$ (with masking as needed) and outputs $\mathbf{O} = \mathbf{P} \times \mathbf{V}$. We emphasize that CA has no trainable parameters (nor gradients) and its per-token computation does not require intermediate outputs from other tokens' CA, but only their $\mathbf{K}, \mathbf{V}$ vectors. The CA compute process is plotted in Figure 1.

At training, storing $\mathbf{P}$ is memory-prohibitive due to its quadratic complexity. Modern IO-aware attention kernels (Dao et al., 2022) avoid materializing $\mathbf{P}$ and recompute it during backward. Hence, CA also generates an negligible amount of intermediate states, rendering it stateless.

2.2 LLM Training Parallelization

Modern LLM training combines four parallelisms – data, tensor, context, and pipeline parallelisms – to scale.

In data parallelism (DP), replicas process different batches and meet at a gradient synchronization barrier. Any replica that receives a batch with a larger attention workload (e.g., from packed data) becomes a straggler and stalls the others.

Tensor parallelism (TP) shards model layers at the cost of per-layer communication, which is not affordable when scaling beyond one node (TP size > 8). TP balances memory and compute across TP shards, as attention is sharded along the head dimension and all devices process exactly the same data batch (Shoeybi et al., 2019).

Context parallelism (CP) shards data along the sequence dimension across multiple GPUs. Context-independent layers run independently on each shard, yet core attention requires an all-gather (AG) to collect token states of its context to compute the attention of the local shard. Under a causal mask, earlier tokens in the sequence performs less attention computation than later tokens, so naively slicing the sequence creates load imbalance. Recent work (Dubey et al., 2024) mitigates this with a head-tail shard assignment: partitions the sequence into $2 \times \mathrm{CP}$ shards and assign rank i both the i-th and the $(2 \times \mathrm{CP} - 1 - i)$-th shard. Further, when document packing is used, *per-document CP* shards each document rather than the entire concatenated chunk to balance workloads, which is discussed in detail in §3.2.

Pipeline parallelism (PP) partitions the model's layers into stages, and splits the input data into microbatches, which flow through each stage concurrently; synchronization occurs when activations are passed between two pipeline stages. To minimize pipeline bubbles, each stage shall ideally have a similar compute workload (Zheng et al., 2022). However, because different microbatches may contain chunks packed with varying lengths of documents, the processing time of each stage become imbalanced, leading to bubbles that propagate downstream that idle all subsequent stages. This imbalance cannot be mitigated by globally resharding the data, as each pipeline stage processes a disjoint set of model layers.

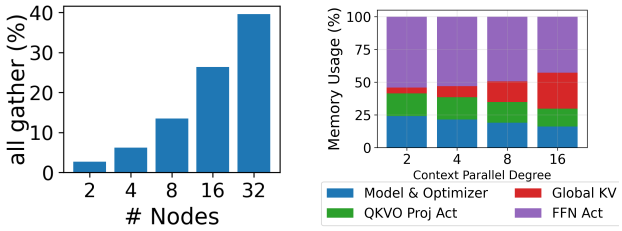
3 CHALLENGE AND MOTIVATION

This section formalizes the problem of compute and memory imbalance under document packing, explains why existing remedies cannot adequately balance both, and motivates core attention disaggregation.

3.1 Load Imbalance in Compute and Memory

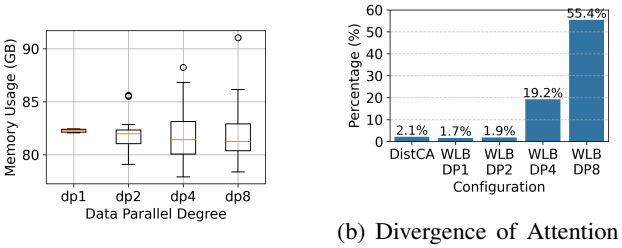
Let the compute of an l-token document be $\mathrm{FLOPs}(l) = \alpha l^2 + \beta l$, where α, β are constants related to LLM architectures (e.g., hidden size). αl^2 represents the core attention FLOPs; βl aggregates FLOPs of context-independent layers. Let activation memory be $\mathrm{M}(l) = \gamma l$, since modern IO-aware attention kernels avoid storing $\mathbf{P}$ in forward and recompute it during backward; activations saved for backward are therefore dominated by context-independent layers.

For a microbatch of documents with lengths $\{l_i\}_{i=1}^n$ packed together, the total compute is $\alpha \sum_{i=1}^n l_i^2 + \beta \sum_i^n l_i$, while the total activation memory is $\gamma \sum_{i=1}^n l_i$. To make two microbatches – of lengths $\{l_i\}_{i=1}^n$ and $\{l'_j\}_{j=1}^m$ – balanced in both compute and memory, both conditions must hold: $\sum_i^n l_i = \sum_j^m l'_j$ and $\sum_i^n l_i^2 = \sum_j^m l'^2_j$, which is difficult to satisfy in practice. Existing methods typically target one condition at a time. For example, fixed-size packing equalizes memory ($\sum_i^n l_i = \sum_j^m l'_j$) by packing the same number of total tokens in both chunks, but leaves attention compute unequal due to the quadratic term.



(a) all-gather latency(%) (b) Memory breakdown (%)

Figure 3. Latency and memory breakdown for all-gather in Context Parallel, Llama-8B. Document lengths are all 32k.



(a) Memory divergence. (b) Divergence of Attention Computation.

Figure 4. Throughput and memory divergence for variable-length data chunks, measured on 512K-token data chunks, 8B model.

Two remedies, variable-length data chunk and per-document context parallelism, adjust the document packing scheme or sequence sharding to approximate these conditions (Wang et al., 2025c), respectively. Next, we quantitatively analyze them and show their inadequacies.

3.2 Problems of Existing Methods

Variable-length data chunk. This approach redistributes documents across microbatches to equalize $\sum_i^n l_i^2 = \sum_j^m l_j'^2$ (FLOPs), i.e., it tries to move several documents from a microbatch with more attention compute to compensate other microbatches. However, token counts $\sum l$ then diverge across microbatches, inflating activation memory on some ranks. Figure 4a shows per-microbatch memory divergence growing with DP size (fixing TP=8), when we scale global batch size proportionally with the number of nodes to keep memory fully utilized; with a 512K maximum length, achieving compute balance requires 1.08 - 1.17x more activation memory on some ranks.

Worse still, as sequence length grows, the method hits the memory cap, where simply moving sequences fails to fully equalize attention compute due to memory constraints. The resulting straggler effect is visible in Figure 4b, where we quantify how much GPU compute is underutilized due to attention imbalance by measuring the percentage of average idle time to average iteration time; the idle fraction rises to 19% at DP=4 and 55% at DP=8 for a 512K-length workload. In such regimes, variable-length chunking is fundamentally

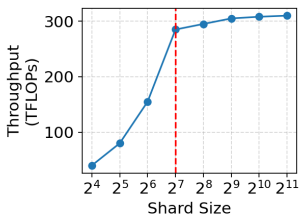


Figure 5. CA Throughput.

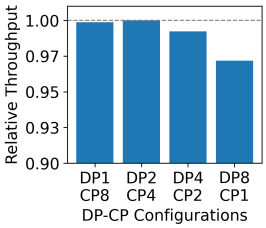


Figure 6. Throughput with variable-length data chunk and per-doc CP.

constrained by memory pressure and becomes ineffective.

Per-document context parallelism. For a chunk packed with documents $l_1, l_2 \dots l_n$, evenly slicing the concatenated chunk yields imbalance because early tokens perform less work under causal masking (even with "head-tail" paring in §2.2). Per-document CP instead *partitions each document* into $2c$ shards (c is the CP degree) and assigns to rank i the i-th and $(2c-1-i)$-th shard of each document. Each CP rank hence processes an equal share of every document – balancing both compute and memory. Despite this, we find that per-document CP encounters three bottlenecks at scale.

First, for ordinarily short documents, per-document CP still partitions them and results in tiny shards. These tiny shards may underfill attention tiles and lower arithmetic intensity (Wang et al., 2025c). Figure 5 shows that the kernel's throughput drops significantly for document shorter than 128 tokens. Second, in per-document CP, the each rank runs $1/c$ of the overall computation, but requires all-gathering the whole KV states, whose cost is linear to the global number of tokens. As shown in Figure 3a, when scaling up CP degree, the all-gather latency share rises from merely 3% on 2 nodes to nearly 40% on 32 nodes. Finally, the last CP rank must store the entire document's aggregated KV states as input states, and release the memory until backward. The memory overhead scales with CP degree. As shown in Figure 3b, the KV memory fraction grows from 3% at 2 nodes to almost 30% at 16 nodes. These bottlenecks fundamentally limit the scalability of per-document CP.

Combination of both. Systems combining these techniques inherit the drawbacks of each as scale grows. Figure 6 shows an experiment on a 64-GPU 512K-token workload. Scaling the CP degree reduces imbalance, but also decreases throughput and risks OOM as batch size or number of nodes increases. On the other hand, increasing DP causes severe load imbalance and suboptimal throughput. This trade-off becomes more acute with larger scale and longer context.

3.3 Motivation for Core Attention Disaggregation

The balancing conditions in §3.1 suggest a natural boundary: separate CA's quadratic term $\alpha \sum l_i^2$ from the linear terms $\beta \sum l_i$ and $\gamma \sum l_i$ associated with context-independent lay-

ers. This boundary is motivated by two key asymmetries. First, *divergent scaling*: CA computation grows quadratically with document length and is stateless (no trainable parameters, no stored activations), whereas context-independent layers grow linearly and are stateful (they store parameters and activations for backward). Second, *incompatible balancing requirements*: applying fine-grained sharding uniformly to all layers – as in per-document CP – assigns each device a different number of tokens. This inherently sacrifices memory balance to achieve compute balance.

These asymmetries motivate disaggregating CA so that it can be parallelized with a different strategy than the rest of the model - something existing fine-grained CP approaches fundamentally cannot achieve. Once CA is decoupled, we can: (1) independently schedule and equalize CA compute across a shared pool of devices, i.e. *attention servers*, without concerning memory, as CA is stateless; (2) independently balance the compute and memory of context-independent layers, which are both linear with tokens. Making this practical hinges on two questions: **(Q1)** how to balance CA compute? **(Q2)** can we hide the communication caused by disaggregation? We next explain two observations that make both possible.

Divisibility and kernel composability. We observe that core attention computation is divisible at the token granularity. Given $\mathbf{Q}$ for target tokens and $\mathbf{K}, \mathbf{V}$ for their context, each shard computes independently. In modern attention kernels, each GPU thread block is assigned a tile of the core attention computation. The kernel can sustain high MFU on variable-length fused sequence, provided its size is larger than this tile. In other words, kernel throughput depends primarily on the aggregate tokens in a fused call, not on their document of origin. This implies, in practice, documents can be arbitrarily sharded then recombined into a single high-occupancy CA kernel (with proper masking) without hurting kernel efficiency.

We corroborate this by profiling FA2 on a 32K-token chunk packed of document shards of varying lengths and context sizes. Within each chunk, we fix the shard length while randomly sampling the context size for each shard; we sample multiple chunks to measure the average throughput. Results in Figure 5 show a high throughput as long as each document shard has more than 128 tokens, which is the kernel tile size set by FA2. Shards shorter than 128 tokens are padded, which waste compute of their assigned thread blocks. This characteristic enables us to solve **Q1** by arbitrarily partitioning any document into shards with a multiple of 128 tokens and recombine them and issue as a single attention kernel, which balances compute without compromising kernel efficiency.

Communication cost. Like CP, disaggregating CA also requires communicating $\mathbf{K}, \mathbf{V}$ of context tokens. We show that this communication has a low overhead for four reasons. First, due to casual mask, the $\mathbf{K}, \mathbf{V}$ of later shards are not needed by earlier shards. We use an all-to-all operation to send only the required shards; Second, when scheduling shards onto devices, the more communication intense shards – the later shards of a document that have more context tokens – can be dispatched on different devices to avoid a straggler in the all-to-all; Third, thanks to the flexibility of arbitrary sharding, we can further optimize the sharding schemes to minimize communication, without being restricted by uniform sequence splits or a fixed CP degree. We develop this algorithm in detail in §4.2; Finally, communication sending the input of a batch's core attention can be overlapped with the computation of another batch; we realized this as a ping-pong execution scheme in §4.1.

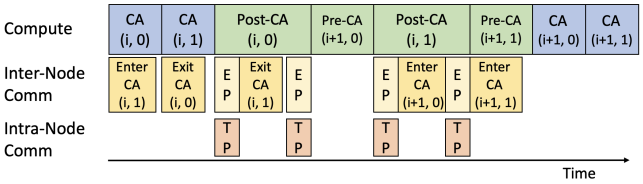


Figure 7. Ping-Pong execution on a single GPU acting as an in-place attention server. Pre-CA denotes computation before core attention (qkv-proj, layer norm, positional embedding); Post-CA denotes computation after core attention (o-proj, FFN, auxiliary operations); EP/TP denotes expert/tensor parallelism communication. Each microbatch is split into two nano-batches (Ping and Pong); $(i, 0)$ is the Ping half of layer i, $(i, 1)$ is the Pong half. Computation of one nano-batch overlaps with communication of the other.

We formally estimate the communication overhead in Appendix A, and show that with a Llama-3-34B's configuration and InfiniBand bandwidth, documents can be partitioned into up to 31 shards and distributed across different devices without incurring communication overhead, and for larger models, this upper bound even increases.

4 METHOD

We implement core attention disaggregation in a system named DistCA. DistCA has two main components: a runtime system and a workload scheduler as shown in Figure 2. The runtime (§4.1) alternate between executing the context-independent layers and the core attention of each transformer layer, inserting the necessary communication between the GPUs responsible for these two parts. The workload scheduler (§4.2) takes a batch of documents as input, and determines how to shard each document and where to place each shard across devices.

4.1 Runtime

In CAD, core attention is computed by a pool of attention servers. More formally, we use *core attention task* to describe attention server's workload. A core attention task (CA−task), denoted by t, is defined as the core attention computation of a query shard $q(t)$ and its context's Key-Value shard $kv(t) = \text{context}(q(t))$. Assume a document is split into multiple non-overlapping shards $q_1, q_2, \ldots, q_n$, the whole document's core attention result is the collection of the corresponding $t_1, t_2, \ldots, t_n$.

As illustrated in Figure 2, in a group of GPUs processing multiple documents, each GPU can function as an independent attention server. For a batch of documents $B = \{d_0, d_1 \ldots d_n\}$, after being processed by context-independent layers, they are split into CA−tasks: $T = \{t_0^{d_0}, t_1^{d_0} \ldots t_0^{d_1}, t_1^{d_1} \ldots\}$. Each CA−task is assigned to an attention server. Assume a server s is assigned a set of tasks, $T_s \subseteq T$. It first receives all inputs of the CA−tasks it is assigned: Receive Tensor $= \{q(t), kv(t) | t \in T_s\}$. Upon receiving all input tensors, due to the divisibility discussed in §3.3, the server can batch all CA−tasks and executes within a single kernel (e.g. via a FlashAttention call). After that, the output of each CA−task is sent back to the originating GPU that handles next context-independent layers.[1]

A central scheduler, running on the CPU, determines the sharding strategy: how to shard and generate all CA−tasks, then assigns each CA−task to an attention server, so the scheduler outputs:

$$T, \{T_s | s \in \text{CA Servers}\} = \text{Scheduler}(B, \text{CA Servers}).$$

We detail the scheduling algorithm in §4.2. While GPUs processing the current batch, the scheduler prefetches documents for the upcoming batch. Using pre-computed profiling data, the scheduler estimates the computational cost for each document and potential CA−tasks, and generates a sharding and assignment plan for the batch.

Incorporating CAD into an end-to-end training system involves several key design choices, which we describe next.

In-place attention server. One straightforward design of attention server is to allocate a dedicated pool of GPUs solely for the CA computation. Although the design is feasible, we find that this design leads to significant memory underutilization for long-context training. As shown in Figure 3b, the FFN layers account for the majority of memory consumption due to their large hidden states; conversely, the core attention is stateless. Therefore, dedicating a separate group of GPUs to core attention would leave their memory largely unused, while the GPUs processing context-independent layers would remain memory-constrained. In-place attention server solves this problem by allowing each GPU to periodically switch its role between computing context-independent layer and acting as an attention server. This switching behavior is identical with or without pipeline parallelism: each GPU simply alternates between its CA-server role and its context-independent-layer role within each transformer layer. By doing so, it achieves both high memory utilization and balanced compute across GPUs.

[1]CAD does not introduce additional nondeterminism beyond standard GPU training: it does not alter the dataloader output order or the mathematical result of core attention, since each document's shards are reassembled identically regardless of execution schedule.

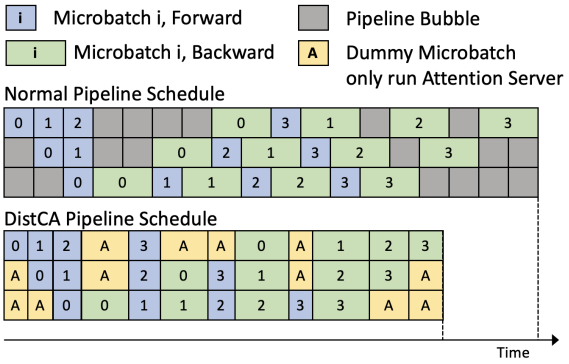


Figure 8. Pipeline Parallel schedule: standard 1F1B (top) vs. disaggregated attention (bottom). Each row is a pipeline stage. Gray slots arise from data dependencies inherent to pipeline parallelism. DistCA fills these idle slots with *dummy microbatches* that compute core attention tasks from other pipeline stages' real microbatches. Since CA is stateless (no model weights or stored activations are needed), these dummy microbatches incur no additional memory overhead.

Ping-Pong execution. To hide communication overhead, we employ a ping-pong execution schedule. Figure 7 illustrates the execution flow on a single GPU. We fuse the post-CA computation of the previous transformer layer with the pre-CA computation of the current layer. This fusion is possible because both sets of operations are context-independent. Each input microbatch is divided into two smaller nano-batches, "Ping" and "Pong", of the same number of tokens. The execution of these two nano-batches is interleaved, allowing the communication of one to overlap with the computation of the other. Furthermore, we overlap the intra-node communication for Tensor Parallel (typically over NVLink) with the inter-node communication caused by core attention disaggregation (typically over InfiniBand).

Pipeline parallelism support. CAD naturally integrates with DP and TP, and replaces CP. We discuss how to integrate CAD with pipeline parallelism. In PP (Figure 8), a training iteration is divided into multiple logical ticks. During each tick, different pipeline stages process different microbatches concurrently. For CA, as it has no weights, CA tasks from different PP stages are indistinguishable from CA tasks across microbatches in DP, both of which will

be scheduled and balanced across attention servers. For context-independent layers (excluding CA), since micro-batches across all PP stages contain the same number of tokens, their corresponding computations will have identical – hence balanced workload.

To prevent device from idling when switching roles between attention server and computing context-independent layers, we adjust the schedule so that all stages perform the same phase within a tick – either all forward or all backward. We realize this by logically deferring selected backward micro-batches to the pipeline bubbles at the end of the schedule, as illustrated in Figure 8, without increasing the number of ticks per iteration. Additionally, during the pipeline warm-up and drain-down phases, some stages are inevitably idle; we repurpose these idle GPUs time for attention servers to running CA tasks. This integration is compatible with 1F1B, the interleaved 1F1B, and other widely-adopted schedules.

4.2 Communication-Aware Greedy Scheduling

The scheduler must balance two competing goals: load balance and communication efficiency. Partitioning and rebatching CA tasks improves balance but might incurs transfers of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$. We therefore solve a constrained optimization problem: (1) minimize load imbalance across attention servers (measured in FLOPs), while (2) minimizing communication volume (measured in bytes).

Profiler. To estimate the cost of a shard's `CA-task`, we build a profiler that benchmarks CA over a grid of query and key-value lengths kv. For each grid point we record ground-truth latency and throughput. Given a `CA-task` t, we predict its execution time by bilinear interpolation over the four nearest grid points. If t lies in the saturation region (i.e., the kernel is at peak throughput), we derive execution time from the max measured throughput instead.

Scheduling units. To formally describe our scheduling algorithm, we introduce *Item* as either a complete document or a shard of a document. The shard can be of any size, as long as it is a multiple of the attention kernel implementation's block size (128 in our case). Each Item already resides on the device that computes its context-independent layers. An Item's CA computation exactly maps to a `CA-task`.

The scheduler input is a batch of Items B and the number of attention servers n (we assume each attention server is identical). The scheduler decides if an Item should be split into smaller Items, and which attention server each Item is assigned to. For PP, the scheduler is called for each logical tick; otherwise, the scheduler is called once per microbatch.

The scheduling algorithm unfolds in the following steps:

1. Determine target load. First, the scheduler computes the ideal per-server load ($\bar{F}$) by summing the total FLOPs of all Items divided n. With $\bar{F}$, attention servers are partitioned into surplus (load $> \bar{F}$) and deficit (load $< \bar{F}$), and sorted by descending deficit.

2. Iterating through deficit servers for migration. For each deficit destination d in order, we attempt to migrate from surplus sources to close d's gap. To find the most efficient Item to migrate for d, the scheduler evaluates each Item candidate using a *cost-benefit heuristic*. First, for each Item, its maximum number of FLOPs to be migrated, denoted as $\Delta F_{\max}$, is determined by the Item's own FLOPs (F_{Item}), the sender's surplus (S_{source}), and the recipient's deficit ($D_{\text{destination}}$): $\Delta F_{\max} = \min(F_{\text{Item}}, S_{\text{source}}, D_{\text{destination}})$. To transfer each shard, there is a communication cost. We choose the shard whose FLOPs equal $\Delta F_{\max}$ but with the minimal communication. We also estimate the communication cost V_{comm} associated with this migration (see Appendix B). With $\Delta F_{\max}$ and V_{comm}, the scheduler calculates a *priority score* for each candidate Item, defined as the communication cost per unit of computation transferred: $E = \frac{\Delta F_{\max}}{V_{\text{comm}}}$. A higher E signifies a more efficient migration. The scheduler selects the Item with the highest score.

Based on the calculated $\Delta F_{\max}$, the selected Item is either entirely migrated if $\Delta F_{\max} = F_{\text{Item}}$, or spitted into two sub-Items, if $\Delta F_{\max} < F_{\text{Item}}$. The newly created sub-Item with $\Delta F_{\max}$ flops is dispatched to the destination attention server, while the remainder is retained by the source server.

3. Termination Condition The scheduler dynamically balances the workload until when the load on each server is within $\epsilon\bar{F}$ (tolerance $epsilon$), or when remaing moves fail to improve E beyond a threshold. In this way, the scheduler ensures the system-wise load balance while avoids unnecessary communication caused by insignificant migration.

5 Implementations

We implemented DistCA with 2K lines of Python code. For efficient dispatching of attention server's input and output, we implemented an All-to-All communication kernel following the idea of (Lei et al., 2025), with another 1K CUDA and C++ code. The communication utilizes NVSHMEM (NVIDIA Corporation, 2025).

Since DistCA only changes the logic of attention computation, we integrate DistCA to Megatron-LM (Shoeybi et al., 2019) to reuse its efficient implementation for token-independent layers, model architecture, 4D parallelization, and end-to-end training pipeline. The integration takes 1k lines of Python code for custom attention layer, ping-pong computation, and pipeline parallelism.

Table 2. Experiment model configurations. "Hidden" is the hidden dimension size, "#Head" is the number of attention heads, and "Head Size" is the per-head dimension.

Name	#Layer	Hidden	#Head	Hdim	GQA
Llama-3-8B	32	4096	32	128	8
Llama-34B	48	8192	64	128	16

6 EXPERIMENTS

6.1 Setup

Model and Hardware. We test DistCA on LLaMA 8B and LLaMA 34B. The model configurations are in Table 2. All experiments are running with NVIDIA DGX H200 nodes. Each node has 8× 140GB H200 GPUs.

Parallelization. As discussed in §2.1, TP offers load balance among ranks, and has negligible communication overhead when the communication is within a device. So we fix TP=8 in our experiments. For PP, we grid search best configurations that avoid out-of-memory(OOM).

After determining TP and PP degrees, we grid search DP and CP degree for the baseline methods. For DistCA, documents are placed sequentially: each device computes a fixed number of tokens for context-independent layers. If a device reaches its token threshold before a document is fully placed, the remaining portion of that document is put to the next device.

Input data. For each experiment, given a number of tokens per batch, we sample 30 batches from an input distribution and report the average throughput. Our experiments use two (synthetic) distributions: (1) Pretrain with upsampled long context documents ("Pretrain"). Following the common practice (Fu et al., 2024), we upsample long documents in a pretrain data distribution by randomly filtering out documents shorter than a threshold. (2) ProLong (Gao et al., 2025): a public dataset specific for long context training. This work shows that a mixture of long and short documents brings the best performance. Compared to "Pretrain", "ProLong" has a higher percentage of long documents.

We vary the maximum number of tokens of a document ("MaxDocLen") to evaluate the system's performance with different context window size. The total number of tokens is determined by the memory capacity. In all test cases, the baselines go out of memory before DistCA, and the total number of tokens for all systems are set to that value.

Baselines. We compare DistCA against Megatron (Shoeybi et al., 2019), FlexSP (Wang et al., 2025b) ("FlexSP+"), and WLB-LLM (Wang et al., 2025c) ("WLB-ideal"), making a few modifications to each baseline to ensure the comparison is fair to them. For Megatron, we employ per-sequence sharding for context parallelism and balance different DP ranks by reordering microbatches to minimize the theoretical FLOP disparity across ranks. FlexSP uses ILP to decide the CP degree of each CP group, and which document is assigned to each group. To mitigate the system overhead and enable tensor parallel, we reimplement FlexSP within Megatron using the FlexSP parallelism planner described in Wang et al. (2025b). We run the solver offline with a 300-second timeout to compute the optimal document placement. For WLB-LLM, since the official implementation is unavailable, we reproduce its adaptive CP sharding policy within our system by sweeping over DP–CP degrees and reporting the best-performing configuration. We omit the deferred execution mechanism (Algorithm 1 in(Wang et al., 2025c)) as it modifies the training dynamics; integrating it is left to future work.

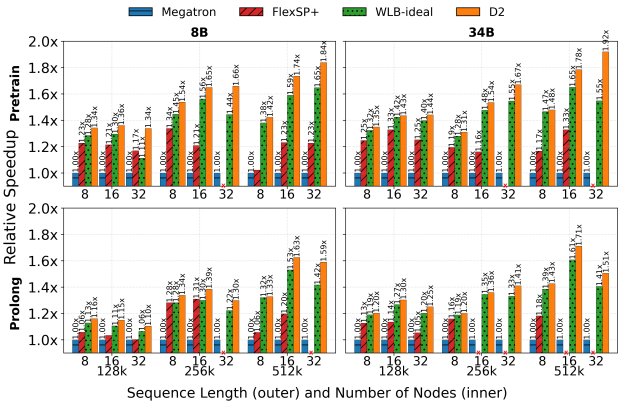


Figure 9. 3D Parallel (no PP) experiment. Relative speedup is defined as the average duration over baseline

We show the relative throughput across DistCA and other baselines in this section. To see the comparison with absolute throughput, see § D.

6.2 End-to-End Experiment

3D parallelism (w/o PP). We first show the performance without PP. Table 4 (in §C) shows the "MaxDocLen", the batch size, and number of GPUs for each experiment.

The result is shown in Figure 9. DistCA consistently outperforms all three baselines, with up to 1.92x speedup against Megatron, 1.35x against "FlexSP+", and 1.20x against "WLB-ideal". For Megatron, the main performance bottleneck mainly comes from attention imbalance, which results from imbalance from data parallel group and from per-sequence CP. The phenomenon gets exaggerated as the number of nodes and "MaxDocLen" increases. For "FlexSP+" and "WLB-ideal", the performance gap stems from the inherent CP–DP trade-off: larger CP increases the CP-related communication and memory overhead, whereas larger DP requires more complex scheduling and balancing across DP ranks, as discussed in §3.2. In contrast, DistCA dispatches

only a subset of tokens and effectively overlaps communication with computation via a ping-pong execution scheme. The fine-grained token-level scheduling space further improves load balance. Moreover, as noted in §3.2, "FlexSP+" and "WLB-ideal" are also constrained by two conflicting factors on the memory side: (1) variable-size data chunks across DP ranks exacerbate memory imbalance, while (2) context parallelism introduces significant all-gather communication overhead. These conflicting factors jointly limit the attainable efficiency of the optimal solution, leading to suboptimal performance in practice.

"FlexSP+" also exhibits additional issues that further degrade performance. First, under the 300-second solver timeout, the ILP solver often fails to find the true optimal sequence placement as the problem size grows. As a result, we observe suboptimal placements in "FlexSP+" that can degrade performance by 1.4x - 3x depending on the solver time budget. Second, "FlexSP+" may still group smaller documents together with larger ones within a single CP group to improve efficiency, which inadvertently increases both memory consumption and communication overhead, resulting in out-of-memory (OOM) failures in certain cases.

With 34B model, DistCA achieves greater speedups at higher "MaxDocLen". This is because a larger "MaxDocLen" leads to a more diverse document length distribution, making it increasingly difficult for "FlexSP+" and "WLB-ideal" to balance the workload effectively.

Interestingly, with 8B model, DistCA achieves greater speedups at lower "MaxDocLen". This is because we use the same number of nodes to train the 8B model as the 34B model. As a result, the total number of tokens per batch is larger, increasing the likelihood of having multiple documents with similar lengths. In this setting, the all-gather overhead becomes more significant: a smaller "MaxDocLen" leads to lower FLOPs per token in the core attention computation, while the all-gather cost remains the same because the total number of tokens is the same. We also observe that DistCA has a higher speedup on "Pretrain" than "ProLong", mainly because "Pretrain" contains more short documents and is more challenging for "FlexSP+" and "WLB-ideal" to balance the workload.

4D parallelism. We now present the performance results under full 4D parallelism. Table 5 (in §C) shows the "MaxDocLen", the batch size and number of GPUs for each experiment. In each setting, we sweep all DistCA, Megatron and WLB across all possible configurations, using a sufficiently large batch size for the given model and workload. To ensure a fair comparison for Megatron and WLB in 34B model, we increase the number of GPUs only for the 34B model to 16, 32, and 64, thereby enlarging the pipeline parallelism search space.

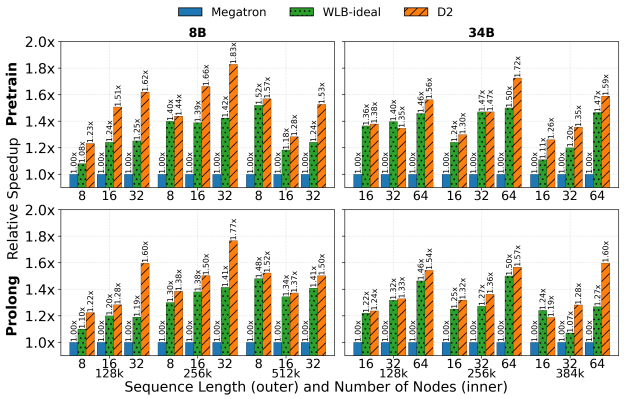


Figure 10. 4D Parallel (with PP) experiment. Relative speedup is defined as the average duration over baseline.

The results are shown in Figure 10. DistCA generally outperforms Megatron and "WLB-ideal" and demonstrates more favorable scaling characteristics over the set of different configurations. For the 8B model (left 2 figures in Figure 10), DistCA achieves up to 1.35x speedup compared to "WLB-ideal" and up to 1.83x speedup compared to Megatron across different configurations. In addition to the advantages described with the 3D parallel (non PP) experiments, DistCA further balances computation across pipeline stages, and repurpose idle stages during warmup and drain-out phases as core attention servers. In contrast, Megatron and WLB-LLM struggles to find effective configurations, often running out of memory at high CP or DP degrees, and experiencing amplified load imbalance caused by pipeline parallelism, which further skews attention computation across stages.

For the 34B model, DistCA also shows positive speedup across most configurations (16, 32, and 64 GPUs), achieving up to 1.25x speedup compared to "WLB-ideal" and 1.72x speedup compared to Megatron. The performance gap generally widens as the maximum document length increases, since larger input diversity exacerbates load imbalance.

Despite these gains, we observe that memory fragmentation creates some runtime overhead that limits DistCA 's performance in the 34B 4D-parallel experiments. Because Core Attention handles requests with varying tensor shapes at each micro-batch, the allocator repeatedly creates and releases differently sized memory blocks, causing fragmentation and frequent PyTorch garbage collection. The resulting CPU overhead delays GPU kernel launches and degrades overall performance. We plan to address this issue in future work with static memory allocation and CUDA Graphs.

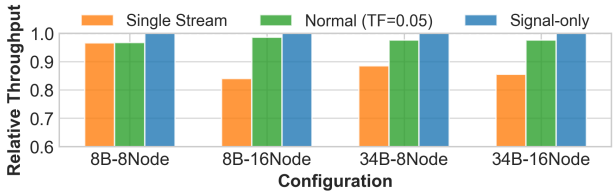


Figure 11. Throughput for different communication patterns.

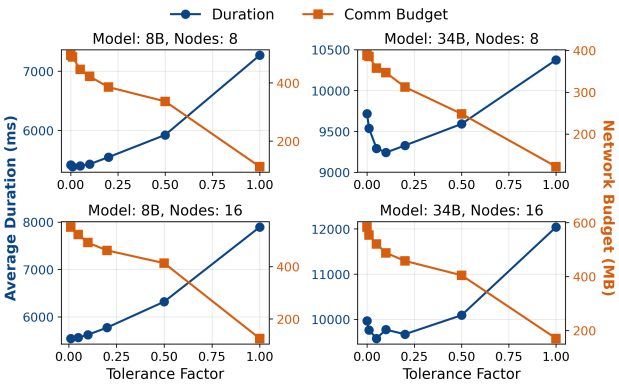


Figure 12. Impact of the compute imbalance tolerance factor.

6.3 Ablation study

To study the effects of different components in DistCA, we designed several ablation studies:

System overhead. The primary system overhead in DistCA lies in the communication volume between a token's attention server and the device for its context-independent layers. We designed two baselines to illustrate the overhead:

1) Signal communication ("Signal"). This reduces each communication volume to 1 byte, meaning the synchronization overhead now only reflects computation imbalance.

2) Remove Ping-Pong execution ("Single Stream"). This makes the communication on the same GPU stream as computation, removing the optimization to overlap the two parts.

Figure 11 reports results on 8B and 34B models across 8 and 16 nodes using the Pretrain distribution, with the total number of tokens fixed to saturate compute. DistCA achieves nearly the same latency as "Signal", indicating that communication is almost fully overlapped with computation, and therefore eliminating the 10-17% higher latency that would otherwise be incurred by "Single Stream".The only exception is the 8B model on 8 nodes, where the compute workload is too small to fully hide communication.

Hyper-parameter in scheduler. The scheduler introduces a tolerance factor that trades off CA load balance against communication volume to minimize latency. Figure 12 shows this trade-off by evaluating 8B and 34B models on 8 and 16 nodes under the Pretrain distribution, using 1M and 512K total tokens respectively with a maximum document length of 128K. For the 8B model, latency remains largely unchanged when the tolerance factor is between 0 and 0.20 as the communication can still be fully overlapped with computation. For the 34B model, however, setting the tolerance factor below 0.10 is too restrictive – communication volume increases and can no longer be hidden, causing higher latency. Conversely, when the tolerance factor is too large, latency roughly linearly rises again due to load imbalance.

Figure 12 also plots the relation between communication size and imbalance tolerance factor. In most cases, tuning tolerance factor from 0 to 0.15 decreases the memory requirement by 20-25% while leaving average duration staying almost the same or – in some cases as shown in 34B, 8 nodes – better. A tolerance factor beyond this point will significantly increase the iteration latency, while leaving the communication size relatively stable.

Further detailed analysis quantifying the system overheads of disaggregation – including QKV splitting/merging cost, theoretical communication bandwidth requirements, and millisecond-level scheduler complexity – are provided in Appendix E.

7 Related Works

Load imbalance in long-context training. Lin et al. (2025) identifies the document length skew across microbatches as a primary cause of slowdowns in today's LLM training systems. Several strategies have been proposed to mitigate this issue. FlexSP (Wang et al., 2025b) introduces dynamic context parallelism. It uses an integer linear programming (ILP) solver to dynamically choose CP degree for device subgroups and to assign documents to these subgroups. However, the ILP is NP-hard, which limits scalability; besides, it only targets a narrower setting (FSDP). Several other approaches also attempt to mitigate the CP overhead either by dynamic CP degree for each document (Ge et al., 2025) or by overlapping communication with attention computation (Liu et al., 2024a; Chen et al., 2025). Compared to these variants of CP, CAD achieves a more precise load balance by a token granularity variable length sharding. CAD also reduces the communication at scale by not only preventing sharding short document, but sharding long documents aware of communication efficiency.

In addition to per-document CP, WLB-LLM (Wang et al., 2025c) introduces variable-size data chunk, which uses an imbalanced MLP to compensate the workload imbalance from attention. Because activation memory is tied to MLP computation, equalizing total FLOPs with this method leads to memory imbalance. As the context length increases, the fraction of MLP computation shrinks, making it harder to

offset attention imbalance.

ByteScale (Ge et al., 2025) performs data-aware CP/DP sharding (named HDP) to reduce redundant communication for short documents. However, it keeps attention and non-attention layers co-located and preserves the conventional execution unit (evenly sharded documents). In this sense, HDP can be viewed as a workload-heterogeneity-aware variant of CP that dynamically adjusts CP degrees. A key constraint is that "different HDP ranks handle an equal number of tokens" (Ge et al., 2025); in practice, device groups with a higher CP degree handle longer documents whose per-token FLOPs are higher due to the quadratic core attention cost, making global workload balance hard to achieve. In contrast, CAD explicitly disaggregates core attention from the rest of the model and assigns *different parallelization strategies* to the two components, allowing attention workload to be balanced independently at token granularity. To benchmark against this family of adaptive CP methods in §6, we compare against FlexSP (Wang et al., 2025b), which formulates workload-heterogeneity-aware parallelism as a per-batch ILP optimization and serves as a strong baseline.

Model disaggregation. Disaggregation, most commonly prefill-decode disaggregation, has been widely adopted in LLM inference (Zhong et al., 2024; Patel et al., 2024; Hu et al., 2025; Qin et al., 2025). The most closely related efforts MegaScale-Infer (Zhu et al., 2025) and concurrent work (Wang et al., 2025a) further disaggregate attention and FFN onto separate physical devices in inference. These work primarily targets mixture-of-expert (MoE) models as the per-layer transfer caused by disaggregation can be merged with the token routing communication inherent to MoE, thereby avoiding additional latency; for dense models, they would incurs extra overhead. In contrast, our work focuses on *language model training*, which is throughput-oriented, and demonstrates benefits independent of model architectures.

8 Limitations

DistCA implements in-place attention servers to maintain memory utilization (§4.1). If memory demand is satisfied, dedicating more GPUs to attention (without scaling those for others) could further reduce compute time while preserving load balance and low communication overhead. We also believe that DistCA could enable better fault tolerance and performance isolation, which we leave to future work.

Our scheduler restricts each `CA-task` to a $\mathbf{Q}$ shard with the full $\mathbf{K}, \mathbf{V}$ context. Allowing a `CA-task` to use a $\mathbf{Q}$ shard with only a sub-range of its $\mathbf{K}, \mathbf{V}$ context would add flexibility. In addition, when estimating communication, the current model pessimistically assumes all tokens are transferred and ignores $\mathbf{K}, \mathbf{V}$ already resident on the destination; this can overestimate bytes and yield non-minimal transfers.

Regarding our evaluation, since public datasets rarely reflect real-world long-context training practices, it currently relies on representative public and synthetic distributions (e.g., upsampled pretrain data (Fu et al., 2024; Wang et al., 2025c) and ProLong (Gao et al., 2025)); testing DistCA on proprietary, industry-scale data remains an open direction. Additionally, as observed in §6, our evaluation at larger model scales is bottlenecked by occasional memory fragmentation overhead, which could be mitigated via static memory allocation in the future.

9 Conclusion

This paper presents core attention disaggregation, a new architecture for large language model training that separates the core attention from the rest of the model to enable independent scaling and scheduling. We observe that core attention is stateless and composable at token granularity, enabling efficient disaggregation with minimal communication overhead. We implement core attention disaggregation in DistCA. DistCA features a workload-aware scheduler to balance computation while minimizing communication, and a ping-pong execution scheme to hide communication latency. End-to-end evaluation shows that DistCA improves throughput by up to 1.92x speedup over Megatron, and 1.35x speedup over existing state-of-the-art training systems with load balancing, with increasing advantages at larger scales.

References

Chen, C., Chen, T., Duan, J., Zhu, Q., Wang, Z., Hu, Q., Sun, P., Li, X., Yang, C., and Hoefler, T. Zeppelin: Balancing variable-length workloads in data parallel large model training. *arXiv preprint arXiv:2509.21841*, 2025.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Rozière, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C.,

Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I. M., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., and et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL https://doi.org/10.48550/arXiv.2407.21783.

Fu, Y., Panda, R., Niu, X., Yue, X., Hajishirzi, H., Kim, Y., and Peng, H. Data engineering for scaling language models to 128k context. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 14125–14134, 2024.

Gao, T., Wettig, A., Yen, H., and Chen, D. How to train long-context language models (effectively). In Che, W., Nabende, J., Shutova, E., and Pilehvar, M. T. (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 7376–7399. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.acl-long.366/.

Ge, H., Feng, J., Huang, Q., Fu, F., Nie, X., Zuo, L., Lin, H., Cui, B., and Liu, X. Bytescale: Communication-efficient scaling of llm training with a 2048k context length on 16384 gpus. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pp. 963–978, 2025.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Hu, C., Huang, H., Xu, L., Chen, X., Wang, C., Xu, J., Chen, S., Feng, H., Wang, S., Bao, Y., Sun, N., and Shan, Y. Shuffleinfer: Disaggregate llm inference for mixed downstream workloads. *ACM Trans. Archit. Code Optim.*, 22 (2), July 2025. ISSN 1544-3566. doi: 10.1145/3732941. URL https://doi.org/10.1145/3732941.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *CoRR*, abs/2309.14509, 2023. doi: 10.48550/ARXIV.2309.14509. URL https://doi.org/10.48550/arXiv.2309.14509.

Lei, Y., Lee, D., Zhao, L., Kurniawan, D., Kim, C., Jeong, H., Kim, C., Choi, H., Yu, L., Krishnamurthy, A., Sherry, J., and Nurvitadhi, E. FLASH: fast all-to-all communication in GPU clusters. *CoRR*, abs/2505.09764, 2025. doi: 10.48550/ARXIV.2505.09764. URL https://doi.org/10.48550/arXiv.2505.09764.

Li, M., Andersen, D. G., Smola, A., and Yu, K. Communication efficient distributed machine learning with the parameter server. *Advances in neural information processing systems*, 27, 2014.

Lin, J., Jiang, Z., Song, Z., Zhao, S., Yu, M., Wang, Z., Wang, C., Shi, Z., Shi, X., Jia, W., Liu, Z., Wang, S., Lin, H., Liu, X., Panda, A., and Li, J. Understanding stragglers in large model training using what-if analysis. In Zhou, L. and Zhou, Y. (eds.), *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pp. 483–498. USENIX Association, 2025. URL https://www.usenix.org/conference/osdi25/presentation/lin-jinkun.

Liu, H., Zaharia, M., and Abbeel, P. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024a. URL https://openreview.net/forum?id=WsRHpHH4s0.

Liu, T., Xu, C., and McAuley, J. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024b.

NVIDIA Corporation. NVSHMEM: GPU-Centric OpenSHMEM. https://developer.nvidia.com/nvshmem, 2025. Accessed: 2025-09-30.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, I., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019.

Qin, R., Li, Z., He, W., Cui, J., Ren, F., Zhang, M., Wu, Y., Zheng, W., and Xu, X. 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), pp. 155–170, 2025.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

Wang, B., Wang, B., Wan, C., Huang, G., Hu, H., Jia, H., Hao, N., Li, M., Chen, N., Chen, S., Yuan, S., Xie, W., Song, X., Chen, X., Yang, X., Zhang, X., Yu, Y., Wang, Y., Zhu, Y., Jiang, Y., Zhou, Y., Lu, Y., Li, H., Hu, J., Lo, K. M., Huang, A., Jiao, B., Li, B., Chen, B., Miao, C., Lou, C., Hu, C., Xu, C., Yu, C., Yao, C., Lv, D., Shi, D., Sun, D., Huang, D., Hu, D., Pang, D., Liu, E., Zhang, F., Wan, F., Yan, G., Zhang, H., Zhou, H., Wu, H., Guo, H., Chen, H., Zhang, H., Wu, H., Zhang, H., Yan, H., Lv, H., Wei, H., Zhou, H., Wang, H., Wang, H., Li, H., Zhou, H., Wang, H., Guo, H., Wang, J., Gong, J., Xie, J., Zhou, J., Sun, J., Wu, J., Zhang, J., Liu, J., Cheng, J., Luo, J., Yan, J., Yang, J., Hou, J., Zhang, J., Cao, J., Yin, J., Liu, J., Huang, J., Lin, J., Tan, K., Li, K., An, K., Lin, K., Liu, K., Yang, L., Zhao, L., Chen, L., Shi, L., Tan, L., Lin, L., Zhang, L., Chen, L., Huang, L., Shi, L., Gu, L., and Chen, M. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *CoRR*, abs/2507.19427, 2025a. doi: 10.48550/ARXIV.2507.19427. URL https://doi.org/10.48550/arXiv.2507.19427.

Wang, S., Wang, G., Wang, Y., Li, J., Hovy, E., and Guo, C. Packing analysis: Packing is more appropriate for large models or datasets in supervised fine-tuning. *arXiv preprint arXiv:2410.08081*, 2024.

Wang, Y., Wang, S., Zhu, S., Fu, F., Liu, X., Xiao, X., Li, H., Li, J., Wu, F., and Cui, B. Flexsp: Accelerating large language model training via flexible sequence parallelism. In Eeckhout, L., Smaragdakis, G., Liang, K., Sampson, A., Kim, M. A., and Rossbach, C. J. (eds.), *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, pp. 421–436. ACM, 2025b. doi: 10.1145/3676641.3715998. URL https://doi.org/10.1145/3676641.3715998.

Wang, Z., Cai, A., Xie, X., Pan, Z., Guan, Y., Chu, W., Wang, J., Li, S., Huang, J., Cai, C., Hao, Y., and Ding, Y. WLB-LLM: workload-balanced 4d parallelism for large language model training. In Zhou, L. and Zhou, Y. (eds.), *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pp. 785–801. USENIX Association, 2025c. URL https://www.usenix.org/conference/osdi25/presentation/wang-zheng.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Zhu, R., Jiang, Z., Jin, C., Wu, P., Stuardo, C. A., Wang, D., Zhang, X., Zhou, H., Wei, H., Cheng, Y., Xiao, J., Zhang, X., Liu, L., Lin, H., Chang, L., Ye, J., Yu, X., Liu, X., Jin, X., and Liu, X. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism. *CoRR*, abs/2504.02263, 2025. doi: 10.48550/ARXIV.2504.02263. URL https://doi.org/10.48550/arXiv.2504.02263.

A UPPER BOUND FOR CORE ATTENTION SERVER MAX PARTITION SIZE

Table 3. Llama-34B configuration

hidden (h)	key-value hidden (h_{kv})	intermeidate (i)
8192	2048	22016

Let a document of length l be divided into s shards. We denote the hidden size of query and key-value tensor as h_q and h_{kv}. The query states for the entire document must be distributed, resulting in a total communication volume of $l \cdot h_q$. Besides, the first key-value shard serves as context of all query shards (0 to $s-1$), the second serves the context of query shards 1 to $s-1$... Assuming the document is evenly sharded, the total communication volume for key-value states is: $h_{kv} \cdot (l/s \cdot s + l/s \cdot (s-1) + \dots) = (s+1)lh_{kv}/2$.

Let the network bandwidth be B, and the time to compute a token's Context-Independent layer be t, When communication is fully overlapped with computation, there should be $t \cdot l \geq l \cdot (h_q + h_{kv}(s+1)/2)/B$. Rearranging the terms gives an upper bound on the number of shards: $s \leq 2(tB - h_q)/h_{kv} - 1$.

We use Llama-34B as the example to analyze the theoretical max number of partition. Its configuration is listed in Table 3. We assume the InfiniBand bandwidth is 50GB/s, and the MFU for Context Independent layer is 50% of the H200 node (990TFLOPs for FP16).

The total flops to compute a token's Context Independent layer is:

$$\begin{aligned} 2 \cdot (h \cdot h \cdot 2 + h \cdot h_{kv} + h \cdot i \cdot 3) &= 2h(2h + h_{kv} + 3i) \\ &= 1320 \cdot 2^{20} \text{ flops} \end{aligned}$$

Here the first factor 2 is because each pair of elements in matrix multiplication has an add operation and a multiplication operation. The Query and Output tensors each has a mapping from a vector of hidden size to another also of hidden size; the Key and Value tensors each has a mapping from hidden size to key-value hidden size; the Feed-Forward Layer applies a Gated 2-layer MLP, so it has 3 mapping from hidden size to FFN intermediate size, or reversely from FFN intermediate size to hidden size.

As a result, the time to compute these layers are:

$$t = 1320 \cdot 2^{20} \text{ flops}/(50\% \cdot 990 \text{ TFLOPs}) \approx 2.796 \cdot 10^{-6}\text{s} \quad (1)$$

Taking this into the formulation $s \leq 2(tB - h_q)/h_{kv} - 1$, we have:

$$\begin{aligned} s &\leq 2(tB - h_q)/h_{kv} - 1 \\ &= 2(2.796 \cdot 10^{-6}\text{s} \cdot 50\text{GB/s} - 16\text{KB})/4\text{KB} - 1 \\ &\approx 31 \end{aligned}$$

The computation time for context-independent layers, t, scales quadratically with the hidden size h_q. Therefore, for larger models, this upper bound on s even increases.

B COMMUNICATION OVERHEAD FUNCTION

This section discusses the formulation of communication cost function $v(\cdot)$.

For a shard of n_q query tokens and n_{kv} key-value tokens, there is:

$$\begin{aligned} v(\Delta F_{\max}, L_q, L_{kv}, \text{size}_q, \text{size}_{kv}) = \min_{n_q, n_{kv}} &\text{Comm}(n_q, n_{kv}) \\ 0 < n_q &\leq L_q \\ n_q + L_{kv} - L_q \leq n_{kv} &\leq L_{kv} \\ \frac{n_q(2n_{kv} - n_q)}{L_q(2L_{kv} - L_q)} &= \frac{\Delta F_{\max}}{F_{\text{Item}}} \\ \text{Comm}(n_q, n_{kv}) &= n_q \cdot \text{size}_q + n_{kv} \cdot \text{size}_{kv} \end{aligned}$$

Let $\alpha = \frac{\Delta F_{\max}}{F_{\text{Item}}}, \beta = \frac{\text{size}_{kv}}{\text{size}_q}$, there is:

$$\text{Comm}(n_q, n_{kv}) = \text{size}_q(n_q(1+\frac{1}{2}\beta) + \frac{\alpha\beta L_q(2L_{kv} - L_q)}{2n_q})$$

As a result, selecting a shard of $n_q^{\text{opt}} = \sqrt{\alpha\beta L_q(2L_{kv} - L_q)/(\beta+2)}$ results in the minimal communication: $v(\cdot) = \text{size}_q \cdot \sqrt{\alpha\beta(\beta+2)L_q(2L_{kv} - L_q)}$. However, if there is $n_q^{\text{opt}} > L_q$ or the corresponding $n_{kv}^{\text{opt}} < n_q + L_{kv} - L_q$, there is the optimal configuration corresponds to the upper bound of $n_q^{\max} = \sqrt{(L_q - L_{kv})^2 + \alpha L_q(2L_{kv} - L_q)}$, and the minimal communication is the corresponding value.

In practice, we notice that different shards have a diverged MFU. As a result, estimating the Item (or the sub-Item)'s time directly by flops is not precise. However, our profiling result also shows that, if we still follow the practice of the head-tail context parallel, i.e. an Item has both the first i to j tokens and the last i to j tokens, the estimation by flops is still accurate. As a result, we keep using the head-tail context parallel and leave a more precise modeling as a future work. In this way, we redefine n_{kv} as the key-value tokens for the "head" half shard, and the communication cost is modified as:

$$\text{Comm}(n_q, n_{kv}) = n_q \cdot \text{size}_q + (L_{\text{doc}} - (n_{kv} - n_q)) \cdot \text{size}_{kv}$$

This is because that the first i to j tokens need the first j token's KV states, while the last i to j tokens need the $L_{\text{doc}} - i$

token's KV states. By definition, there is $i + n_q = j = n_{kv}$. Hence, the communication equals:

$$L_{\text{doc}}\text{size}_{kv} + \frac{1}{2}\text{size}_q\left(n_q(2+\beta) - \frac{\alpha\beta L_q(2L_{kv} - L_q)}{n_q}\right)$$

As a result, the optimal n_q corresponds to the minimal possible value. Based on other constraints, there is

$$\begin{aligned} n_q^{\min} =& L_{kv} - \sqrt{L_{kv}^2 - \alpha(2L_{kv} - L_q)L_q} \\ v(\cdot) =& L_{doc}\text{size}_{kv} + \\ & \frac{1}{2}\text{size}_q\left(n_q^{\min}(2+\beta) - \frac{\alpha\beta L_q(2L_{kv} - L_q)}{n_q^{\min}}\right) \end{aligned}$$

C Experiment Setup

Table 4. 3D Training Configurations.

Model	MaxDocLen	Batch Size	#GPU
Llama-8B	128K	8, 16, 32	64, 128, 256
Llama-8B	256K	4, 8, 16	64, 128, 256
Llama-8B	512K	2, 4, 8	64, 128, 256
Llama-34B	128K	4, 8, 16	64, 128, 256
Llama-34B	256K	2, 4, 8	64, 128, 256
Llama-34B	512K	2, 4, 8	64, 128, 256

Table 5. 4D Parallel Training Configurations.

Model	MaxDocLen	Batch Size	#GPU
Llama-8B	128K	32, 64, 128	64, 128, 256
Llama-8B	256K	16, 32, 32	64, 128, 256
Llama-8B	512K	8, 8, 16	64, 128, 256
Llama-34B	128K	32, 64, 128	128, 256, 512
Llama-34B	256K	16, 32, 32	128, 256, 512
Llama-34B	384K	8, 8, 16	128, 256, 512

D Experiment Result with Absolute Throughput

In § 6, Figure 9 and Figure 10 shows the relative performance of DistCA compared to baselines.

This section presents the absolute aggregate throughput (Figure 13) and per-GPU throughput (Figure 14) in k-tokens per second under the same setup described in § C.

E System Overheads of Core Attention Disaggregation

QKV splitting and merging overhead. Disaggregating core attention requires splitting QKV tensors before dispatching CA tasks and merging output tensors afterward.

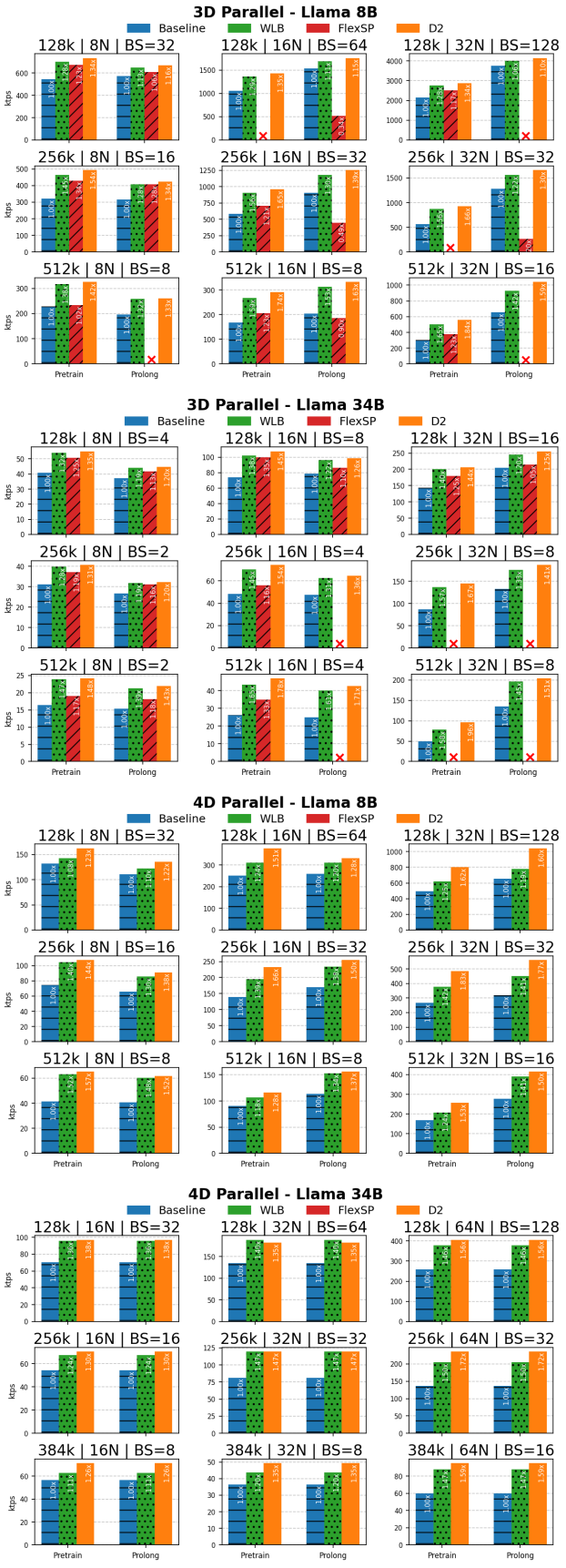


Figure 13. Aggregate absolute throughput (k-tokens/s) across different datasets, model sizes and 3D/4D paralleism.

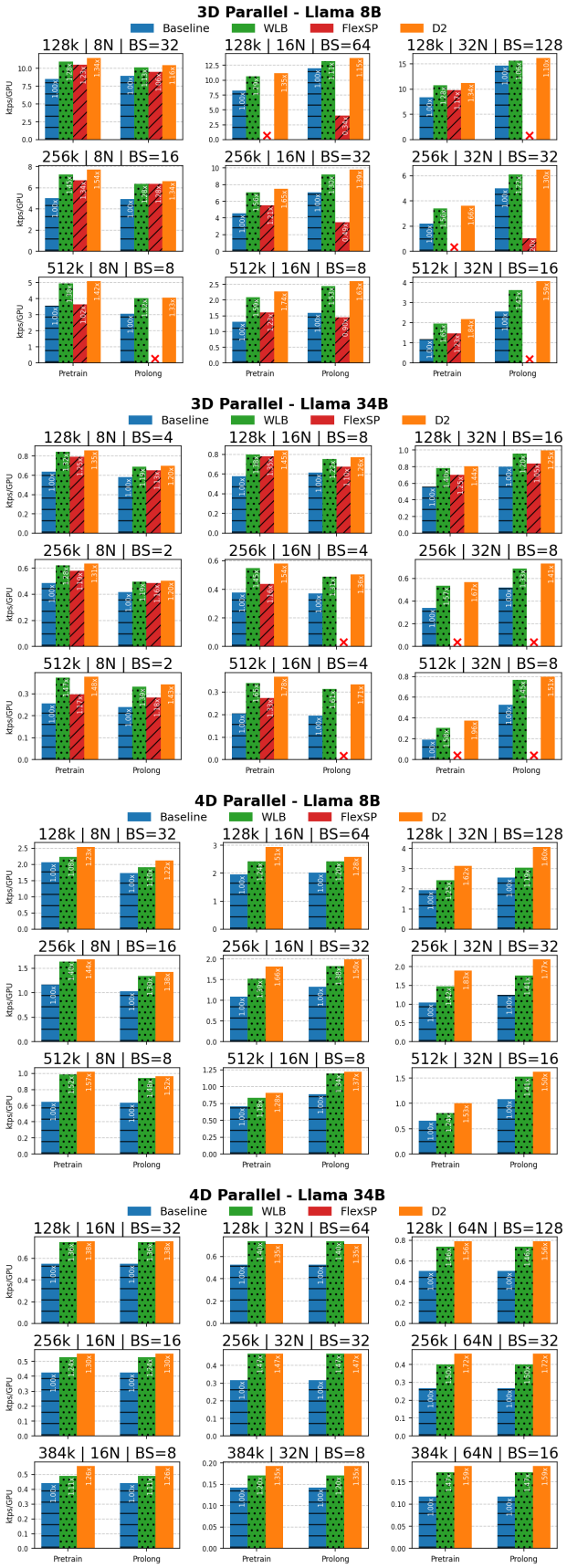


Figure 14. Per-GPU absolute throughput (k-tokens/s) across different datasets, model sizes and 3D/4D paralleism.

Table 6. Overhead of QKV tensor splitting and output merging, measured as percentage of iteration time.

Model	#Nodes	SeqLen	QKV(%)	O(%)	Total(%)
8B	8	128K	1.61	0.54	2.15
8B	16	128K	0.57	0.30	0.87
8B	32	128K	0.43	0.28	0.71
34B	16	128K	0.69	0.32	1.00
34B	32	128K	0.37	0.18	0.55
34B	64	128K	0.28	0.14	0.42

Table 6 quantifies this overhead. The cost ranges from 0.42% to 2.15% of iteration time, and decreases as the cluster scales up (more computation to amortize the fixed overhead). This confirms that the tensor manipulation cost of disaggregation is negligible.

Table 7. Theoretical network bandwidth required to fully overlap CAD communication with computation (512K context). All values are well below the InfiniBand bandwidth available per GPU (50 GB/s).

Model	BS	#Nodes	Tol. Factor	BW (GB/s)
8B	2	8	0.15	41.3
8B	4	16	0.05	40.3
8B	8	32	0.05	39.7
34B	2	8	0.10	33.8
34B	4	16	0.05	33.3
34B	8	32	0.05	32.7

Communication bandwidth analysis. Table 7 reports the theoretical minimum bandwidth to fully hide CAD communication via ping-pong overlap, computed as the ratio of maximum data transfer size to the minimum available computation time. Two observations stand out. First, as model size scales up, the non-CA computation time grows quadratically with hidden size while the QKV transfer size grows only linearly, relaxing the bandwidth requirement (41.3 GB/s for 8B vs. 33.8 GB/s for 34B). Second, under weak scaling, the required bandwidth slightly decreases due to improved kernel efficiency and lower imbalance. All values are well below the per-GPU InfiniBand bandwidth (50 GB/s) available in our cluster, consistent with the near-perfect overlap observed in Figure 11.

Scheduler overhead. The scheduler's complexity is $O(NK)$, where N is the number of documents and K is the number of devices. In our experiments (TP=8), scaling from 64 to 256 GPUs incurs 50–300 ms of scheduling time depending on batch composition. Importantly, the scheduler runs entirely off the critical path: it requires only the document lengths of the next batch, which the dataloader prefetches while GPUs execute the current training step. Since training iteration times are on the order of seconds or longer, the millisecond-level scheduling overhead is fully overlapped.