Context Parallelism for Scalable Million-Token Inference

Amy (Jie) Yang[1], **Jingyi Yang**[1], **Aya Ibrahim**[1], **Xinfeng Xie**[1], **Bangsheng Tang**[1], **Grigory Sizov**[1], **Jeremy Reizenstein**[1], **Jongsoo Park**[1], **Jianyu Huang**[1]

[1]Meta Platforms, Inc.

We present context parallelism for long-context large language model inference, which achieves near-linear scaling for long-context prefill latency with up to 128 H100 GPUs across 16 nodes. Particularly, our method achieves 1M context prefill with Llama3 405B model in 77s (93% parallelization efficiency, 63% FLOPS utilization) and 128K context prefill in 3.8s. We develop two lossless exact ring attention variants: `pass-KV` and `pass-Q` to cover a wide range of use cases with the state-of-the-art performance: full prefill, persistent KV prefill and decode. Benchmarks on H100 GPU hosts inter-connected with RDMA and TCP both show similar scalability for long-context prefill, demonstrating that our method scales well using common commercial data center with medium-to-low inter-host bandwidth.

Date: November 1, 2024
Correspondence: Jianyu Huang at jianyuhuang@meta.com

1 Introduction

Contemporary large language models (LLMs), such as Llama (Touvron et al., 2023a,b; Llama Team, 2024), Gemini (Gemini Team, 2023, 2024), GPT-4 (Achiam et al., 2023), require significant computational resources for inference, especially with long context lengths: OpenAI GPT-4o 128K context length (ope), Anthropic's Claude with 200K context length (ant), Google's Gemini 1.5 Pro with 1M context length (goo). With a single H100 GPU host (8 GPUs), it can take 60 seconds to serve 128K context length[1] or 1200 seconds to serve 1M context length for Llama3 405B model. Context parallelism (CP) is a system optimization technique that improves the latency and scalability of LLM inference, particularly for long contexts. Without modifying the underlying dense attention algorithms, CP offers several advantages for long-context LLM inference:

- **Compute parallelization**: CP distributes computation across multiple GPUs in order to reduce latency, in contrast with pipeline parallelization (PP) (Huang et al., 2019) that improves throughput but not latency.
- **Communication message size reduction**: Compared to tensor parallelism (TP) (Shoeybi et al., 2019), CP demands less communication bandwidth in multi-host environments, by maintaining a communication size that is orders of magnitude smaller than TP, especially for inter-node communication.
- **KV cache distribution**: Key and value (KV) embeddings grow linearly with context length. CP distributes the storage of KV embeddings across multiple GPUs, enabling larger batch sizes with the addition of more CP ranks.

To the best of our knowledge, this is the first paper to disclose the system implementation details on applying context parallelism in inference scenario. Our main contribution lies in the adaptation and optimization of ring attention (Liu et al., 2023) for efficient LLM inference with long context lengths. While the previous work primarily focuses on leveraging ring attention to enhance training throughput for long sequences, this paper identifies and addresses unique challenges posed by inference:

- **Support for multi-turn prefill and decoding**: We recognize the importance of multi-turn conversations, a common characteristic of online LLM applications. Unlike prior research focused on training, we introduce

[1]Google's Gemini 1.5 Pro (Sep 2024) has a latency of 20.43 seconds for time to first token on 100K context length, from https://artificialanalysis.ai/models/gemini-1-5-pro/prompt-options/single/100k.

novel strategies on load-balanced sharding for persistent KV cache and parallelization algorithms that leverage sharded KV cache across multi-turn prefill and decode. These mechanisms are crucial for maintaining conversation history during inference.

- **Optimization for latency**: Latency is critical for user experience in real-time inference. To optimize latency in multi-turn conversations, we developed `pass-KV` and `pass-Q` ring attention variants and heuristics to dynamically select the ring attention algorithms for the lowest latency under varying context lengths and KV cache hit rates.
- **Compute and memory load balancing**: To maintain balanced load among CP ranks across batched requests with varying input lengths, we introduce load-balanced sharding of both input tokens and KV cache entries. Previous work targets training typically with uniform sequence length. We proposed innovative algorithms to ensure even distribution of compute and KV cache memory across CP ranks, contributing to improved overall performance and scalability.

In essence, our work extends context parallelism to efficiently address the challenges and requirements of serving millions of tokens in LLM inference. We introduce novel algorithms and heuristics for optimizing ring attention, demonstrating their effectiveness in reducing latency, improving KV cache utilization, and enabling scalable distributed inference for long-context LLMs. Since our method focuses on system-level optimizations, it can be seamlessly integrated with architectural innovations or algorithmic enhancements to further amplify performance gains.

2 Background

2.1 Large Language Models (LLM)

Since the introduction in the seminal work (Vaswani, 2017), the transformer model architecture has become the fundamental building block for modern language models. Recently, language models have increased exponentially in complexity (measured in number of parameters). Examples: BERT was trained with 0.34B parameters in 2018 (Devlin, 2018), 1.5B parameter GPT-2 was released in 2019 (Radford et al., 2019), and 175B parameter GPT-3 was released one year later in 2020 (Brown, 2020), and the latest Llama 3.1 model pushed to 405B parameters (Llama Team, 2024).

Besides the parameter number, the *context length* is another important indicator of LLM's capabilities. In general, a longer context window indicates better capability to handle a large body of input texts, audios, images, and videos. Modern LLMs support 128K to more than 1M context lengths (ope; ant; goo).

2.2 Challenges with Serving Long Context LLM

In this work, we mainly address the challenges with extremely large (128K-1M) context lengths.

- **Compute**: While an W-parameter Transformer model requires $2 \cdot W$ matrix multiplication FLOPs for each token during inference or forward pass (Kaplan et al., 2020), the pairwise attention architecture found in mainstream transformers (Vaswani, 2017) incurs a quadratic cost in FLOPs w.r.t. context lengths, which would be dominating in long context cases. Several approximate and sparse methods were proposed, including focusing attention on a subset of tokens, and employing a combination of local and global attention strategies. Techniques such as window attention (Liu et al., 2021), local attention (Xiong et al., 2021), Linformer (Wang et al., 2020), and semi-local sparse attention (Jiang et al., 2024; Beltagy et al., 2020) are examples of such innovations that help manage the computational cost.
- **Memory**: Memory usage for LLMs, particularly the KV cache (Pope et al., 2023), scales linearly with the context length. Model compression techniques such as KV cache quantization are crucial for bending the growth curve: lower precision formats like 3-bit, INT4/8 or FP8 can achieve a $2\times$ to $4\times$ reduction in memory requirements compared to using 16-bit (Hooper et al., 2024; Lin et al., 2024). Grouped Query Attention (GQA) (Ainslie et al., 2023) and MQA (Shazeer, 2019) were widely adopted to reduce memory usage by reducing the number of KV heads by $8\times$ to $64\times$. Additionally, strategies like paged

attention (Kwon et al., 2023) have been developed to provide efficient page-like memory management for large numbers of tokens.

2.3 Prior works on Long Context LLM

The following are the main directions to achieve efficient long context window LLM inference:

- **New model architectures**: introduce long context window comprehension components at pretraining stage (Munkhdalai et al., 2024).
- **Post-training changes**: modify a pretrained model with shorter context window to support longer or even infinite context windows (Xiao et al., 2023).
- **System-level optimizations**: preserve the model architecture, instead improve the scalability of existing dense attention algorithms to leverage more compute resources (Li et al., 2021; Brandon et al., 2023; Liu et al., 2023).

Our work falls into the third category, and can be used in conjunction with methods from the other two categories with minor or no modifications. Our method accelerates future algorithmic research or real-world LLM applications for long-context LLM serving, and also provides the flexibility to trade off model inference latency with hardware capacity depending on the latency requirements of specific applications.

3 Context Parallel Inference

3.1 Model Parallelization

Large language models are commonly parallelized across multiple GPUs using a combination of various parallelism paradigms: **Tensor Parallelism (TP)** (Shoeybi et al., 2019; Korthikanti et al., 2023) partitions the weights of fully connected layers (i.e., linear layers) by alternating the sharding in row and column dimensions. **Pipeline Parallelism (PP)** (Narayanan et al., 2021) shards layers into different pipeline stages, and splits input tensors along the batch size dimension into micro-batches to orchestrate a pipeline schedule to optimize the system throughput. Instead of sharding model weights, **Context Parallelism (CP)** (Li et al., 2021) distributes input tokens to multiple GPUs along the sequence length dimension. CP ranks communicate QKV tensors for attention, which is the only computation with dependency between tokens in the same sequence.

Both TP and CP reduce latency when scaled to multiple nodes. Compared with TP, CP provides an alternative design choice for trade-offs between memory consumption and system performance. As detailed in Table 1, CP communicates token embeddings on attention layers while TP communicates on linear layers. CP has less communication traffic for two reasons: (1) Contemporary LLMs have more linear layers than attention layers: each canonical transformer block has four linear layers and one attention layer. (2) CP may communicate KV tensors instead of Q tensors, which leads to much less communication for models with GQA (Ainslie et al., 2023). For Llama3 405B model with 128 query heads and 8 KV heads ($N_{KV} = 8$ vs. $N_H = 128$), communicating KV heads has 16× smaller message sizes than communicating query heads (Llama Team, 2024). CP's communication cost advantage over TP results in significant latency improvements for multi-node inference, as interconnect bandwidth between nodes are several times lower than intra-node bandwidth (Section 4.2.2). Although CP offers lower communication costs, it incurs higher memory consumption because its lack of model weight sharding.

In this paper, we design and implement an efficient LLM inference system with CP to unblock such a trade-off when scaling out the number of GPUs. In practice, we set TP size into a number (usually 8) to fit the model into GPU memory, and we leverage CP to efficiently scale out into multiple nodes as it saves communication traffic.

3.2 Inference Prefill and Decode Attention

We characterize large language model online inference for multi-turn messaging into three stages: full prefill, partial prefill, and decode. When user initiates the conversation with an initial prompt, the entire prompt goes through **full prefill**, where we compute full causal attention between tokens. Projected key and value tensors

Table 1 Communication and memory cost comparison between tensor parallel (TP) and context parallel (CP) for full prefill. T: sequence length, D_H: head dimension, N_H: # of attention heads, N_{KV}: # of key/value heads, N_{TP}: TP group size, W: model parameter size. Total comm cost shows the communication cost per transformer block.

	TP	CP
Collective	AllReduce	SendRecv
Comm per 2 Linear	$T \cdot N_H \cdot D_H$	0
Comm per Attn	0	$T \cdot N_{KV} \cdot D_H$
Total comm	$2 \cdot (T \cdot N_H \cdot D_H)$	$T \cdot N_{KV} \cdot D_H$
Parameter Size	$\frac{W}{N_{TP}}$	W

from multi-head (MHA) or grouped query attention (GQA) (Ainslie et al., 2023) are saved in GPU HBM as KV cache. After the initial full prefill, the model then starts generating a response with auto-regressive **decoding**, where a new token attends to previously cached KV tensors and outputs response tokens one at a time. KV values generated during decoding stage are also saved in KV cache. After the server returns a response, the user may give a follow-up prompt, which will go through **partial prefill** (or **persistent KV prefill**), where tokens within the new prompt attend to themselves as well as all cached tokens in the previous prompt and model response. This process may repeat multiple times in real world applications, which requires persistency of KV cache between prompts from the same user.

3.3 Computation and Communication Modeling

Each of the three stages of multi-turn online LLM inference carries different performance characteristics.

Assume we have an input sequence with length T, with previously cached KV length P, and a generic GQA model with N_H query heads, N_{KV} key and value heads and model dimension D. We have the following shapes for query (Q), key (K), and value (V) embeddings:

$$shape(Q) = [T, N_H, \frac{D}{N_H}]$$

$$shape(K) = shape(V) = [(T + P), N_{KV}, \frac{D}{N_H}]$$

When Q and KV have the same lengths, passing KV around in ring attention incurs smaller traffic than passing Q, and the communication can be fully overlapped with attention computation (Li et al., 2021). LLM training guarantees this property $len(Q) = len(K) = len(V) = T$, or equivalently, $P = 0$. This is not necessarily true for inference as $len(Q)$, $len(K)$, and $len(V)$ depend on user behaviors and KV cache configurations.

For inference, with high persistent KV hit rate, the ring attention algorithm that always passes KV around may not provide the best performance, as:

- Attention computation is much faster with fewer Q than cached KV. Communication cost will be exposed on critical path if not fully overlap with computation.
- When Q is significantly smaller than the cached KV, communicating the full persistent KV would be significantly more costly than communicating Q.

To achieve better inference performance for full prefill, persistent KV prefill, and decode, we extend ring attention with an option to pass Q instead of KV, when passing Q leads to less communication cost. Specifically, Q embeddings have smaller size than KV embeddings if:

$$\frac{T}{T+P} \leq 2 \cdot \frac{N_{KV}}{N_H} \quad (1)$$

Note that the right hand side (RHS) is constant given a pretrained model. Therefore we can use the RHS as a constant threshold to switch between passing KV embeddings and Q embeddings dynamically depending on $\frac{T}{T+P}$, or the *KV cache miss rate* (1− *KV cache hit rate*).

Table 2 GQA attention complexity for full prefill and partial prefill (e: number of bytes per element).

	FULL PREFILL	PARTIAL PREFILL
FLOPS	$4T^2D$	$4TD(T+P)$
Q BYTES	TDe	TDe
KV BYTES	$2TD\frac{N_{KV}}{N_H}e$	$2(P+T)D\frac{N_{KV}}{N_H}e$

Specifically, for full prefill where $P = 0$, communicating KV embeddings results in a smaller message size for GQA models with $N_H > 2 \times N_{KV}$. For decoding where $T = 1$, communicating Q embedding almost always results in smaller communication sizes. Consequently, we leverage ring pass-KV for full prefill, and ring pass-Q for decode and partial prefill with high KV cache hit rate.

To understand whether communication can be reliably overlapped with attention computation with varying persistent KV hit rates, we approximate the attention computation and QKV communication latency using a simple roof-line model.

Let's assume a system with peak compute of C, bandwidth of BW for QKV communication, new token length T, and cached token length P. We focus the analysis on prefill with low persistent KV hit rate, which is compute-bound and the culprit of long (e.g. 60s) prefill latency for inference. In the following analysis, we aim to identify values of P and T such that the communication latency is smaller than the computation latency. In simplified terms: $\frac{FLOPS}{C} \geq \frac{min(Q_{bytes}, KV_{bytes})}{BW}$.

For low-to-medium KV cache hit rate prefill, we will not be bound by ring pass-KV communication if:

$$\frac{4 \cdot T \cdot D(T+P)}{C} \geq \frac{2 \cdot (T+P) \cdot D \cdot e \cdot \frac{N_{KV}}{N_H}}{BW}$$

To extend to multi-host distributed inference, we would further partition each CP rank with TP over intra-node GPUs, and add additional CP nodes to increase parallelization on context dimension. For CP over N nodes, we would be able to hide ring pass-KV communication latency under attention computation if:

$$\frac{4 \cdot T \cdot D(T+P)}{N \cdot C} \geq \frac{2 \cdot (T+P) \cdot D \cdot e \cdot \frac{N_{KV}}{N_H}}{BW}$$

$$T \geq N \cdot \frac{C \cdot N_{KV} \cdot e}{2 \cdot N_H \cdot BW} \tag{2}$$

Note that the threshold for T, the length of new tokens is a static threshold with respect to a given model and hardware, which is independent of KV cache hit P.

Similarly, in a distributed inference setting with CP over N nodes, we will not be bottlenecked by ring pass-Q communication if:

$$\frac{4 \cdot T \cdot D(T+P)}{N \cdot C} \geq \frac{T \cdot D \cdot e}{BW}$$

$$(T+P) \geq N \cdot \frac{e \cdot C}{4 \cdot BW} \tag{3}$$

Note that RHS is also static with respect to one particular system. As we have discussed, we will leverage pass-Q when the number of new tokens to prefill T is significantly smaller than the number of cached tokens P. In this case, whether we will be able to completely overlap the latency for communicating Q is determined by the total context length $(T + P)$. Sufficiently large total context length would allow us to overlap the pass-Q communication regardless of KV cache hit rate.

To summarize, we adaptively switch between pass-KV and pass-Q for inference partial prefill following the heuristics in Algorithm 1[2]. We can calculate the static thresholds for this heuristics once based on the system

[2]In practice, the achieved BW and C are lower than the theoretical hardware peaks. We start with these peak values and then fine-tune the thresholds based on empirical data.

Algorithm 1 Pass-KV vs. Pass-Q Partial Prefill Heuristics

```
if $T \geq N\frac{C \cdot N_{KV} \cdot e}{2 \cdot N_H \cdot BW}$ or $\frac{T}{T+P} \geq 2\frac{N_{KV}}{N_H}$ then
    pass-KV
else
    pass-Q
end if
```

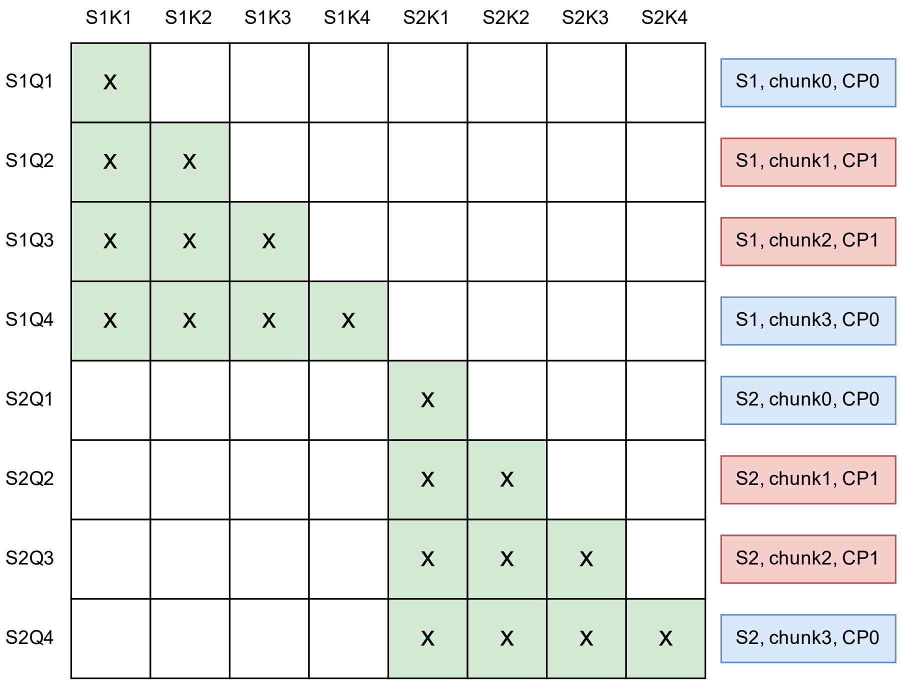


Figure 1 Load-balanced CP sharding with fused inputs in full prefill with 2 CP ranks (CP2). We have 2 input sequences: $S1$, $S2$. Each is partitioned evenly into 4 chunks: Q_i / K_i, where $i = 1, 2, 3, 4$.

and model spec, and use the heuristics to choose which options to use dynamically for the optimal performance in a wide combination of total context length and KV cache hit thresholds.

3.4 Ring Pass-KV, Pass-Q Prefill

We implemented both pass-KV and pass-Q ring attention to minimize the communication latency with different context lengths and KV cache hit rate. In this section, we delve into the implementation details for achieving effective load balancing and communication overhead management, which are critical to the the scalability of distributed context parallel inference.

3.4.1 Load Balanced Sharding

In causal attention each token attends to all tokens before it in the same sequence. Naively partitioning all tokens evenly over CP ranks in the order of the original sequence results in imbalanced compute over different CP ranks. Prior work leverages order permutation and uneven partition to achieve load balance for causal attention (Cho et al., 2024; Brandon et al., 2023). To support maximum context length provided by the pretrained model without OOM on any particular CP rank with heavier load, we aim for load-balancing for both attention compute and KV cache capacity. To shard an input sequence into N CP ranks, we partition the sequence evenly into $2 \times N$ chunks: $C_0, C_1, ..., C_{2\times N-1}$, and have each CP rank i take two chunks: (C_i, $C_{2\times N-i-1}$).

For fused variable length inputs in full prefill, we partition each individual sequence in the same way and pad

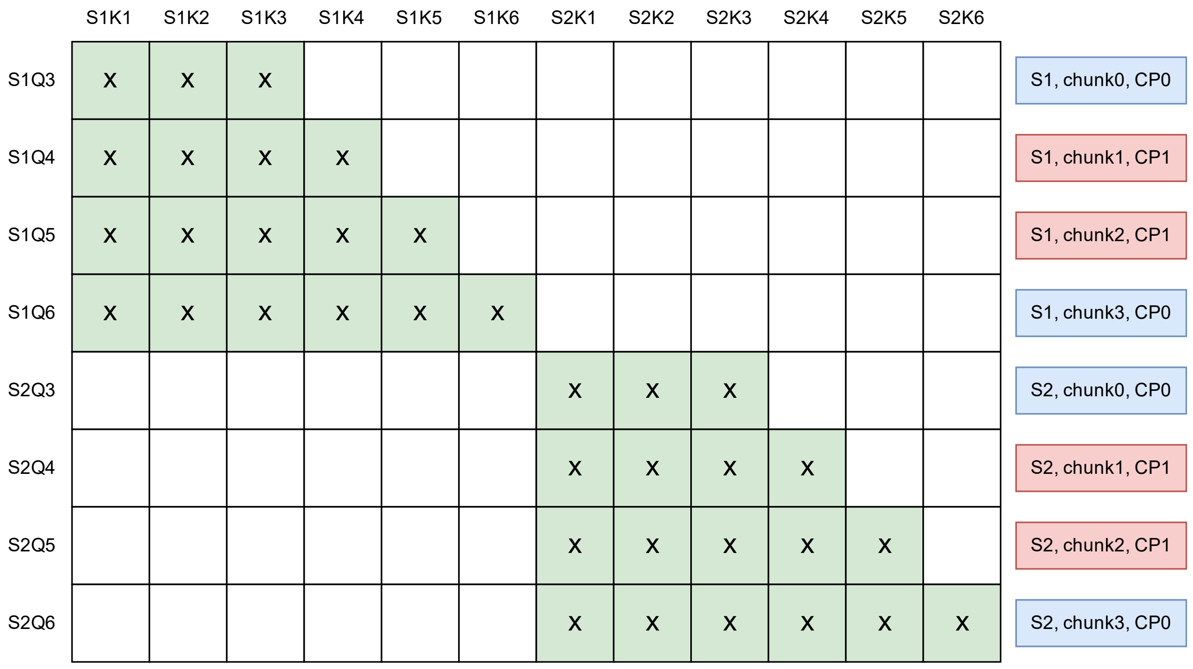


Figure 2 Load-balanced CP sharding with fused inputs partial prefill with 2 CP ranks (CP2). We have 2 input sequences: $S1$, $S2$. Load-balanced sharding is applied to the new token Q_i dimension (4 chunks), regardless of how cached token dimension K_i is partitioned in partial prefill.

the input sequence length if needed (Figure 1).

For partial prefill with new tokens (total length: T) and cached tokens (total length: P), we apply the load-balanced sharding in the dimension of the new tokens regardless of cached tokens (Figure 2).

3.4.2 Ring Pass-KV Algorithm

In Llama3 training (Llama Team, 2024), the all-gather based pass-KV algorithm is utilized, which initially performs an all-gather on the key and value tensors, followed by computing the attention output for the local query tensor chunk. The all-gather communication latency becomes a bottleneck in the critical path, complicating the overlap of operations during inference, especially with variant sequence lengths in a batch and partial prefill used in multi-turn chat. Conversely, the ring-based pass-KV approach, while reducing the computation in smaller granularity, facilitates the overlapping of *SendRecv* with attention computations within the ring loop.

We further make a modification to the ring pass-KV algorithm (Liu et al., 2023) to better suit the partial prefill use case in multi-turn chats. Here an invariant we need to maintain for the ring algorithm is passing equal-sized messages between CP ranks to adhere to collective communication interfaces. CP ranks hold different numbers of KV embeddings as a result of multi-turn chat. Padding and decoding introduce slight variations in KV embedding length per rank even though our load-balanced sharding distributes KV embeddings evenly.

Assume we have N CP ranks $CP_0, CP_1, ..., CP_{N-1}$ with cached KV lengths of $P_0, ..., P_{N-1}$, and partial prefill new tokens of length T. We pass KV embeddings of length $\max_{0 \le i < N}(P_i) + \lceil T/N \rceil$ around CP ranks in a ring (Figure 3), where $\lceil T/N \rceil$ indicates the lengths of load-balanced sharding (Section 3.4.1) of T tokens over N ranks.

For fused variable sequence lengths (Varseq) partial prefill of B sequences in one batch, assume we have sequences $S^0(P^0, T^0), ..., S^{B-1}(P^{B-1}, T^{B-1})$. The i-th sequence S^i has P^i cached KV embeddings, T^i new prefill tokens, with P^i_j cached tokens and T^i_j new tokens sharded to CP rank j. We have Algorithm 2 for a ring pass-KV partial prefill with fused inputs for CP over N hosts. KV^s_k indicates key and value embeddings

Algorithm 2 Fused Varseq Ring Pass-KV Partial Prefill

```
for $i = 0$ to $B - 1$ do
    $L^i \leftarrow max_{0 \le j < N}(P^i_j + T^i_j)$
end for
// On CP rank $k$
$KV^k_k \leftarrow concat^{B-1}_{i=0}(pad(P^i_k + T^i_k, L^i))$
$Q_k \leftarrow concat^{B-1}_{i=0}(T^i_k)$
$p \leftarrow (k - 1) \mod N$
for $j = 0$ to $N - 1$ do
    $s \leftarrow (k - j) \mod N$
    Rank $k$ sends $KV^s_k$ to next rank
    Rank $k$ receives $KV^s_p$ from previous rank
    Compute $O^s_k \leftarrow GQA(Q_k, KV^s_k)$
    $KV^s_k \leftarrow KV^s_p$
end for
Compute $O_k \leftarrow merge^{N-1}_{s=0}(O^s_k)$
```

Algorithm 3 Fused Varseq Ring Pass-Q Partial Prefill

```
// On CP rank $k$ with $KV_k$
$Q_k \leftarrow concat^{B-1}_{i=0}(T^i_k)$
$p \leftarrow (k - 1) \mod N$
for $j = 0$ to $N - 1$ do
    $s \leftarrow (k - j) \mod N$
    Rank $k$ sends $Q^s_k$ to next rank
    Rank $k$ receives $Q^s_p$ from previous rank
    Compute $O^k_s \leftarrow GQA(Q^s_k, KV_k)$
    $Q^s_k \leftarrow Q^s_p$
end for
Permute $\{O^k_s\}^{N-1}_{s=0}$ and All2All to recover $\{O^s_k\}^{N-1}_{s=0}$.
Compute $O_k \leftarrow merge^{N-1}_{s=0}(O^s_k)$
```

received from rank k which is originally allocated to rank s.

In the ring algorithm, Q_k, Q embeddings sharded to rank k, need to attend to all key and value embeddings sharded to all ranks: $KV_0, KV_1, ..., KV_{N-1}$. The attention compute between Q_k and KV_j is overlapped with *SendRecv* for KV_{j-1} from a neighbor rank. We pass KV_j in a ring $N - 1$ times and each rank executes N partial attention compute.

At the end of the ring algorithm loop, each CP rank k will have the attention output of O^s_k with $s = 0, 1, ..., N-1$, where O^s_k denotes the attention output from Q_k and KV^s (key and value embeddings originally sharded to rank s, see bottom of Figure 3). We then apply a merge attention operator (Juravsky et al., 2024) to get the result of Q_k interacted with all KV embeddings across CP ranks (See Appendix C, Equation (4)).

3.4.3 Ring Pass-Q Algorithm

Passing Q embeddings around while keeping K and V embeddings stationary will have partial attention results scattered across CP ranks. We need to have another round of collective communication over CP process group to restore the partial outputs to the original source rank. Following the notations of ring `pass-KV` algorithm in Section 3.4.2, we have Algorithm 3 for ring `pass-Q` attention (Figure 4). Similarly, Q^s_k indicates a Q embedding received from rank k which was initially allocated to rank s. Note that with `pass-Q` we have the guarantee that all CP ranks have the same embedding lengths for query as a result of load-balanced sharding (Section 3.4.1).

All2All for partial attention outputs is on the critical path and therefore introduces an additional communication

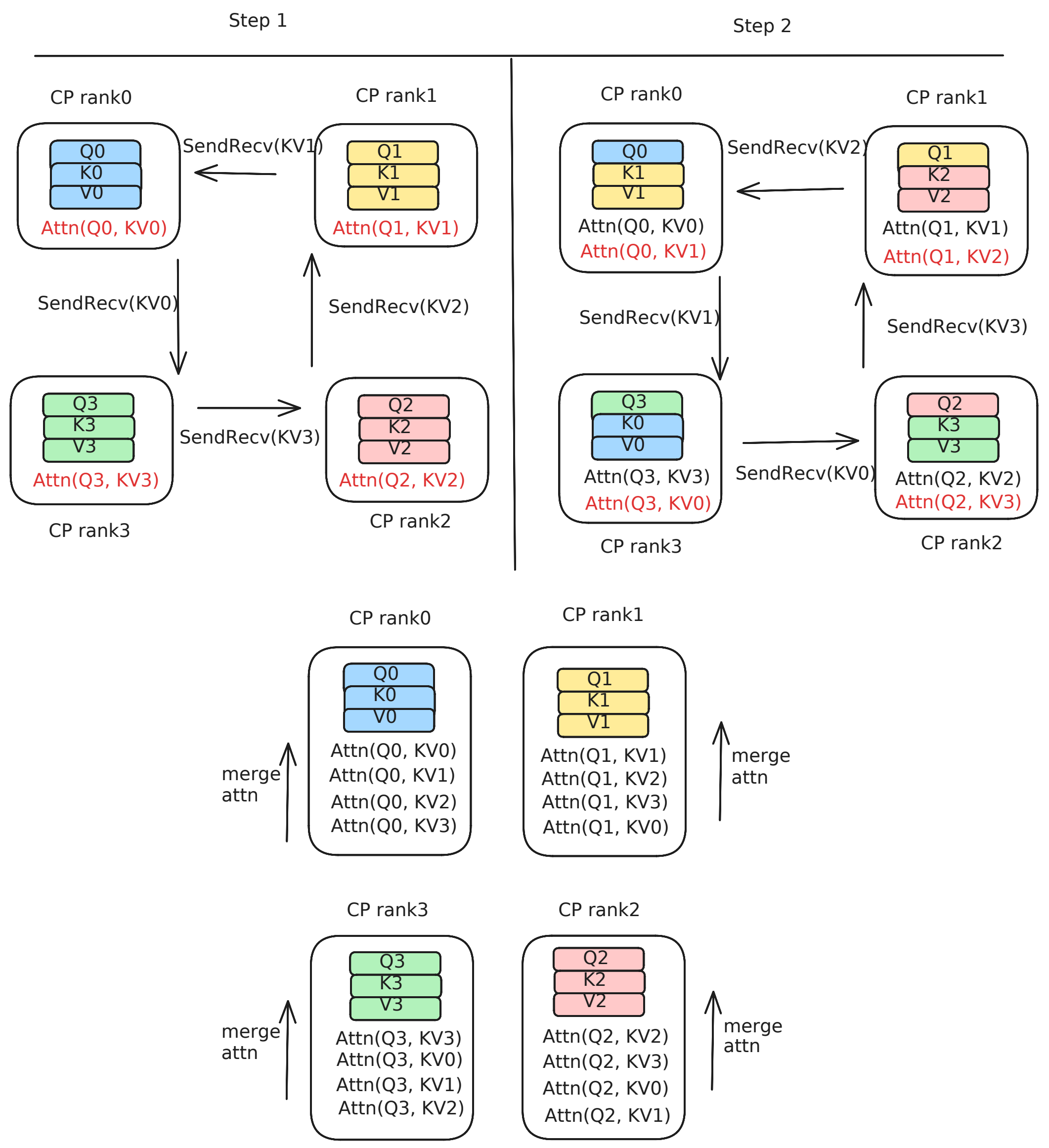


Figure 3 Ring Pass-KV Attention with 4 CP ranks (CP4).

overhead apart from the communication for passing query embedding. The analysis for overlapping query embedding and attention in Equation (2) and (3) only applies to the ring communication. The heuristics in Algorithm 1 for switching between `pass-KV` and `pass-Q` doesn't take *All2All* latency into account[3].

3.5 Ring Pass-Q Decode

With multi-turn prefill and decode, key and value embeddings of the decode tokens are also stored in the KV cache. As decoding generates one response token at a time for each sequence, each decode batch contains exactly one token for each sequence in the batch. If context-parallel decode consistently shards the decoding tokens of a sequence to one specific rank, the rank that handles both decode and prefill will encounter load imbalance issues: it will have longest KV cache and out-of-memory (OOM) before other ranks reach their KV cache capacity.

To ensure we utilize full KV cache capacity from all CP ranks, we implemented *batched ring pass-Q decode*

[3]We present a refined algorithm in Appendix D and provide a detailed time breakdown for validations in Table 4.

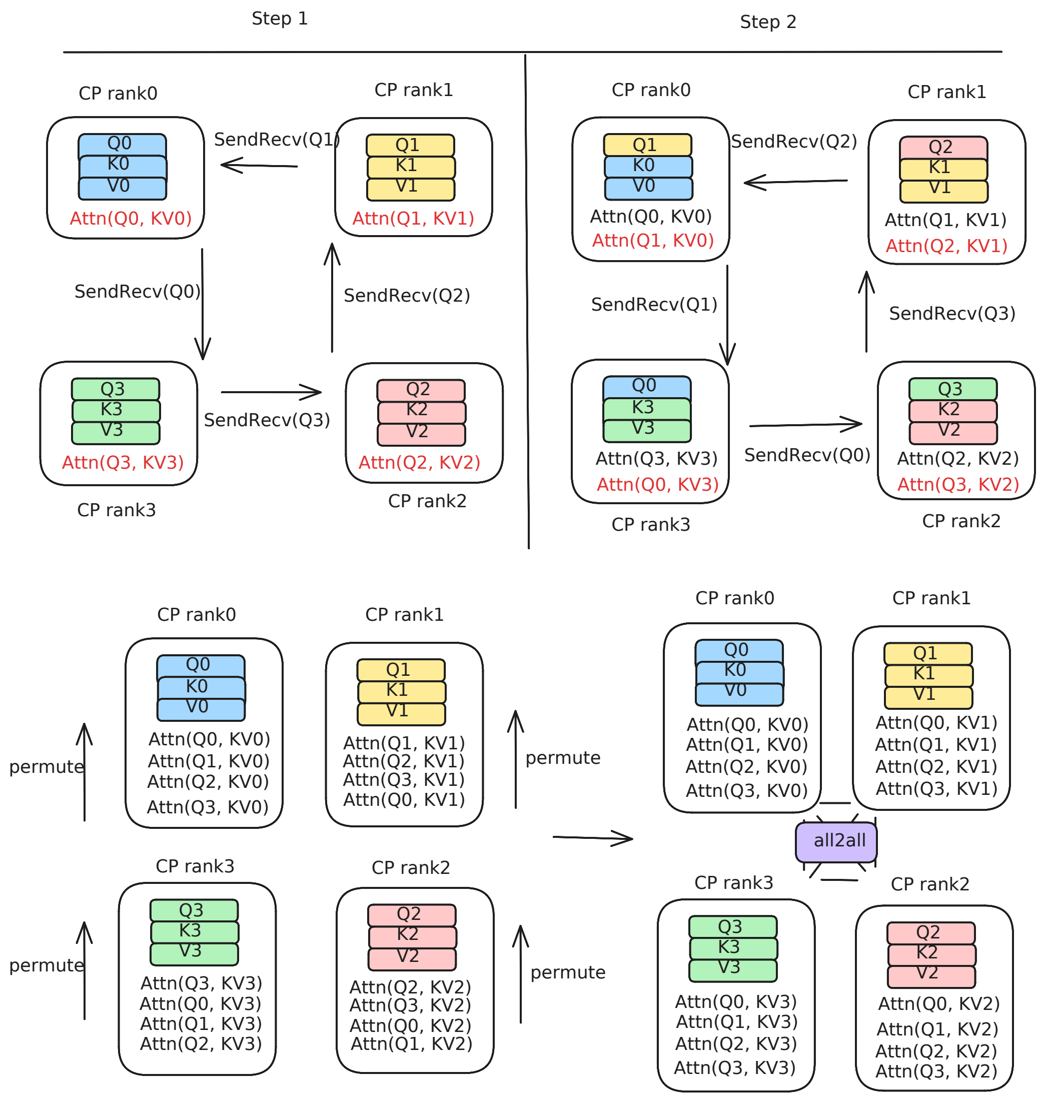


Figure 4 Ring Pass-Q Attention with 4 CP ranks (CP4).

where we offset by 1 index for each decode iterations and shard batched decode evenly with round-robin. With exactly 1 token per sequence for decode, we pass Q rather than K and V embeddings to minimize communication size (Equation 1). Algorithm 4 summarizes our CP decode algorithm with the same notations used for prefill algorithms.

Similar to ring pass-Q prefill, we need to permute the partial attention output order and communicate scattered partial attention outputs back to the original source ranks.

4 Experiments

4.1 Experiment Setup

We used Llama3 405B model with row-wise quantized FP8 weights (Llama Team, 2024) for feed forward layers after GQA. Llama3 405B is a dense transformer model with 126 transformer layers, 16384 model dimension, 128 query heads, and 8 key and value heads (Table 9).

We ran our performance benchmarks on the Grand Teton platform (Meta Engineering, 2022), where each

Algorithm 4 Batched Ring Pass-Q Decode

```
// On CP rank $k$ with $KV_k$, query $Q_k$, batch ids $bid_k$
$p \leftarrow (k-1) \mod N$
for $j = 0$ to $N-1$ do
    $s \leftarrow (k-j) \mod N$
    Rank $k$ sends $Q_k^s$, $bid_k^s$ to next rank
    Rank $k$ receives $Q_p^s$, $bid_p^s$ from previous rank
    Compute $O_s^k \leftarrow GQA(Q_k^s, KV_k[bid_k^s])$
    $Q_k^s \leftarrow Q_p^s$
    $bid_k^s \leftarrow bid_p^s$
end for
Permute $\{O_s^k\}_{s=0}^{N-1}$ and $All2All$ to recover $\{O_k^s\}_{s=0}^{N-1}$.
Compute $O_k \leftarrow merge_{s=0}^{N-1}(O_k^s)$
```

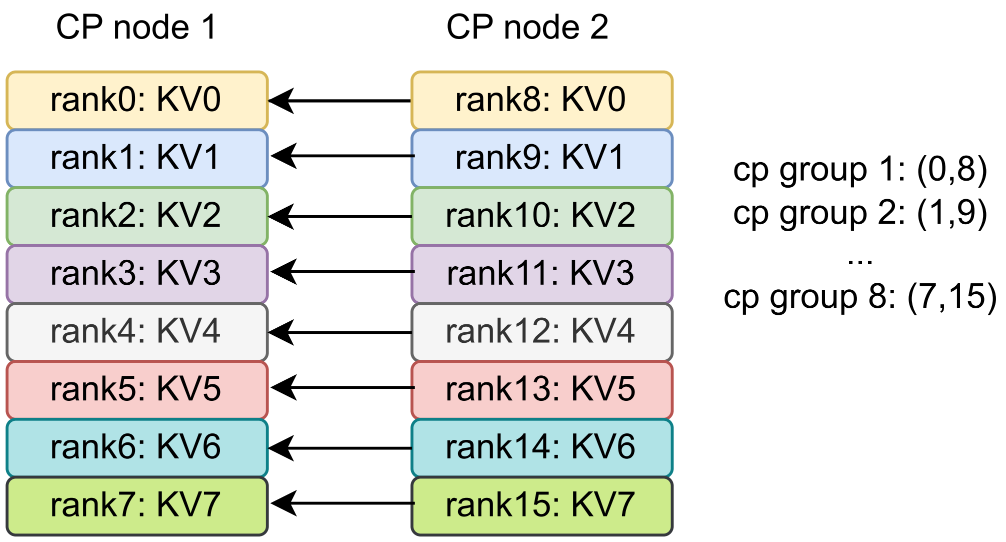


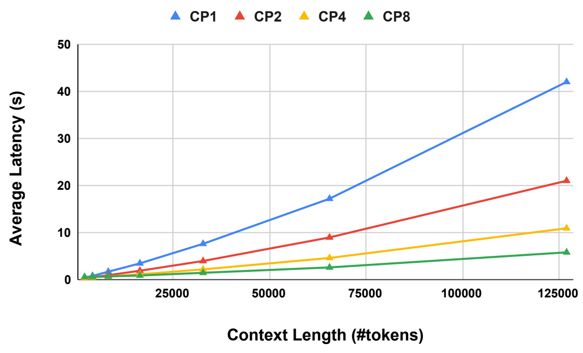
Figure 5 Context parallel across nodes and tensor parallel within nodes, with 2 CP ranks (CP2).

host has 8 Nvidia H100 GPUs fully connected with NVLink ("host" and "node" are interchangeable in the subsequent text). Each H100 GPU is equipped with 96GB HBM2e with 2.4 TB/sec peak memory bandwidth. We tested on two subtypes of Grand Teton platforms: Grand Teton Training (GTT) and Grand Teton Inference (GTI). GTT hosts are inter-connected with backend RDMA network with 400 Gb/s per GPU, and GTI hosts are inter-connected with frontend network over TCP/IP with 100 Gb/s per GPU.

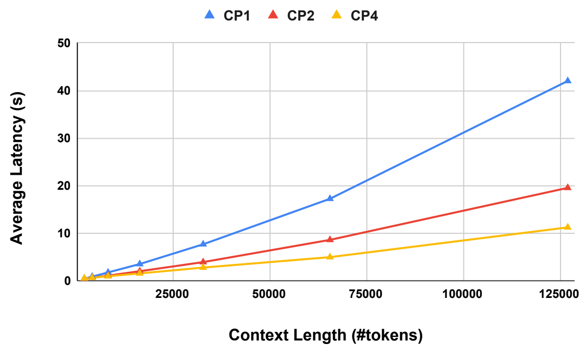
With row-wise FP8 quantization[4], the entire 405B model fits into one node with TP8 (tensor parallelism across 8 partitions) partitioning. Each GPU holds 1 KV head and 16 Q heads, and feed forward layers are partitioned with alternating column and row parallelism (Shoeybi et al., 2019). Flash Attention 3 (Shah et al., 2024) is adopted for attention kernels in prefill, while Flash Decoding (fla) with number of K/V splits 256 is used during decoding.

We tested full prefill, partial prefill, and decode performance with context parallelism over 1-16 nodes. Within each CP node the model is partitioned with TP8 over 8 GPUs. We form one CP communication group per KV head, with each CP group consisting of N GPUs (one GPU in each node) holding the same KV head in their respective tensor parallel groups. Ring communication around CP ranks is implemented an 8-way *SendRecv* (Figure 5).

[4]https://github.com/pytorch/FBGEMM/tree/main/fbgemm_gpu/experimental/gen_ai



(a) GTT Latency for CP with 1, 2, 4, 8 nodes.



(b) GTI Latency for CP with 1, 2, 4 nodes.

Figure 6 Llama3 405B `pass-KV` full prefill latency.

4.2 Context Parallel Prefill Scaling

4.2.1 Latency Reduction with Fixed Context Length

Llama3 405B model supports a maximum of 128K context window, which is equivalent to 300-400 pages of books. We used max batch size 1 and tested how the full prefill latency for context lengths 2K to 128K vary with respect to the addition of more CP nodes.

Figure 6a shows the full prefill latency of `pass-KV` full prefill on GTI and GTT for 1-8 CP nodes. With sufficiently large context lengths, the latency for passing key and value embeddings are overlapped with attention compute, and we get proportional latency reduction with more CP nodes: latency for the same input length is halved as we double the number of CP nodes. Specifically, with CP8 on GTT, an FP8 Llama3 405B model can process a 128K token prefill in 5.85 seconds.

For GTI systems with much lower inter-host bandwidth over frontend TCP/IP network, we observe the same scalability with up to 4 nodes. Inspecting the GPU trace from GTI, we found the achieved bandwidth for inter-host communication is roughly 3GB/s per rank, which is still enough to overlap the `pass-KV` communication with attention compute, demonstrating the robustness of `pass-KV` algorithm even with low inter-connect bandwidth.

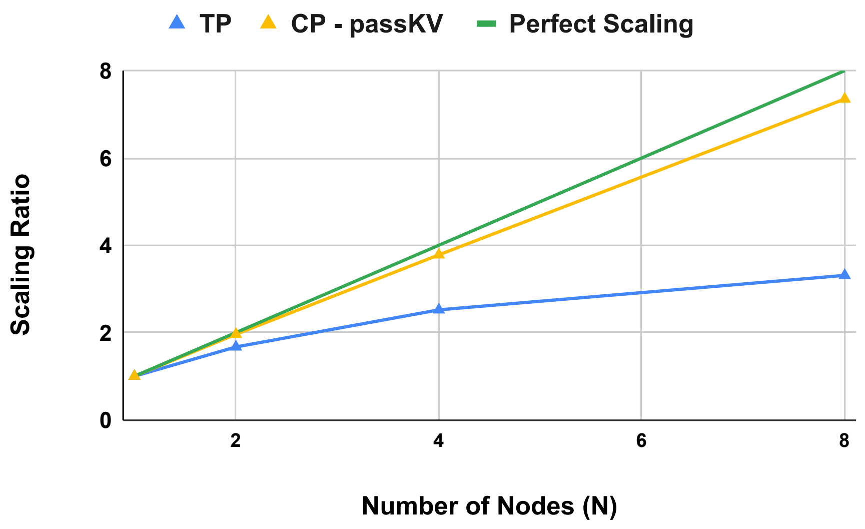


Figure 7 Scaling ratio (latency with one node over latency with N nodes) of context parallel vs. multi-node tensor parallel.

4.2.2 Comparing with Multi-Node Tensor-Parallel

To compare with context-parallel performance, we benchmarked tensor-parallel over multiple nodes on GTT with up to 8 nodes. Llama3 405B model has 8 KV heads. To effectively parallelize 8 KV heads across more than 8 GPUs, we replicate each KV head over N_{TP}/N_{KV} GPUs where N_{TP} is the total number of GPUs in the tensor parallel group and N_{KV} is the number of KV heads. Query heads are distributed evenly to all GPUs with N_H/N_{TP} query heads per GPU. Computation is still fully parallelized over N_{TP} GPUs.

We calculate scaling ratio for a paralellization across N nodes as as τ_1/τ_N, where τ_N is the latency for N nodes to process a 128K context prefill. Better parallelization algorithms would have scaling ratios closer to N.

Figure 7 illustrates the scaling ratios for multi-node tensor parallelism compared to context parallelism across 1 to 8 GTT nodes. Tensor-parallel becomes more bottlenecked by inter-host communication with the growth of capacity, as *AllReduce* latency increased significantly with the addition of more nodes. While the latency is different by roughly 15% between CP2 and TP16 on 2 nodes, the difference drastically increases to 100% when scaled to 8 nodes.

This evaluation is performed on H100 hosts which exhibit significantly lower inter-host bandwidth compared to intra-host badwidth. For future GB200 (nvg) with NVLink connecting multiple hosts, tensor parallelism can still benefits with reasonable scalability.

4.2.3 Scaling Context Length with Fixed Capacity

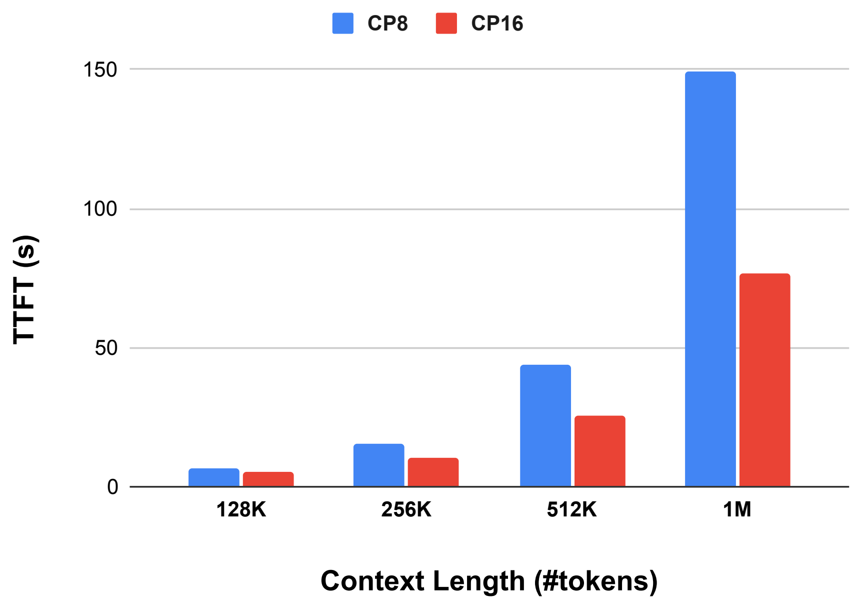


Figure 8 TTFT of 128K-1M context with 8 and 16 CP ranks (CP8 and CP16).

By partitioning the KV cache across CP ranks, we also enhance the KV cache capacity as more CP nodes are added. To demonstrate scalability in terms of both capacity and latency, we run up to 1M context prefill over 8 and 16 GTT nodes. This corresponds to approximately 1 hour of video content. With a 16-node setup, we achieve an exact prefill in 77 seconds for a 1M context length and 3.8 seconds for a 128K context length (Figure 8). The quadratic increase in attention latency with context length begins to dominate the overall time to first token (TTFT) latency, resulting in more than 2× increase in TTFT with a 2× increase in context length for $\geq$ 512K token prefill.

We calculate the FLOPS utilization of a 1M context length on 16 nodes in Appendix B. The achieved FLOPS is 502 TF/sec per H100, compared to a standalone Flash Attention v3 benchmark performance of 540 TF/sec

Table 3 TTFT (in ms) for pass-KV vs. pass-Q varying P and T with $P + T = 128000$, on 4 CP ranks (CP4).

P	T	Miss Rate	pass-KV	pass-Q
126720	1280	1.00%	1023.39	898.71
124800	3200	2.50%	1110.18	1046.43
123840	**4160**	**3.25%**	**1298.92**	**1280.1**
121600	**6400**	**5.00%**	**1305.56**	**1302.01**
115200	12800	10.00%	2080.67	2205.27
102400	25600	20.00%	3353.02	3617.02
89600	38400	30.00%	4629.23	4922.52
76800	51200	40.00%	5745.08	6217.83
64000	64000	50.00%	6845.21	7367.99
51200	76800	60.00%	7890.35	8468.66
38400	89600	70.00%	8697.27	9666.62
25600	102400	80.00%	10105.78	10652.39
12800	115200	90.00%	11136.4	11571.62
0	128000	100.00%	11462.15	12360.57

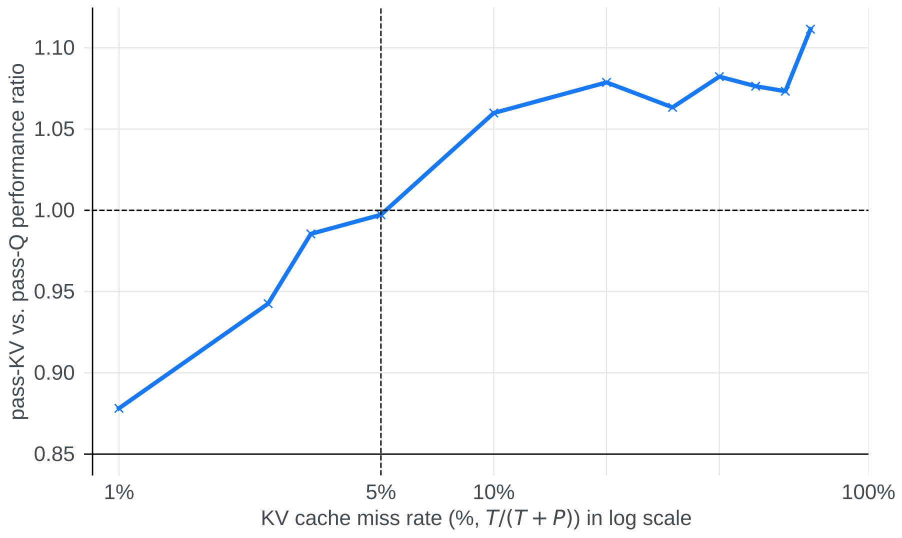


Figure 9 pass-KV / pass-Q speed ratio of 128K context with persistent KV cache miss rate, varying P and T with $P + T = 128000$, on 4 CP ranks (CP4).

for 8K context length (1M over 128 H100 GPUs) on a single GPU, resulting in a 93% parallelization efficiency. Considering the peak FLOPS on the specialized H100 configurations, we achieve approximately 63% FLOPS utilization.

4.2.4 Pass-KV vs. Pass-Q Partial (Persistent KV) Prefill

The persistent KV cache provides substantial advantages in long-context LLM inference by minimizing repeated computational overhead in multi-turn conversations. In Table 3, experiments with a 128K context length on 4 GTT nodes demonstrated that, in both pass-KV and pass-Q implementations, TTFT latency is linearly proportional to the persistent *KV cache miss rate* ($\frac{T}{T+P}$).

Figure 9 compares pass-KV and pass-Q in terms of the KV cache miss rate. When the KV cache miss rate is less than 5%, pass-Q exhibits better latency; however, when the miss rate exceeds 5%, pass-KV achieves lower latency.

The tipping point between pass-Q and pass-KV occurs at $T = 6400$ (5% KV cache miss rate). Table 4 details the time breakdown for cache miss rates slightly below and above this configuration (2.5% and 10% miss rate). *SendRecv* and Attn represent the *SendRecv* time and the partial attention compute time (in μs) for

Table 4 Time breakdown (in μs) on pass-KV vs. pass-Q ring attention at cache miss rate of 2.5% and 10% with $P + T = 128000$, on 4 CP ranks (CP4).

Miss Rate	pass-KV/Q	*SendRecv*	Attn	*All2All*
2.5%	pass-KV	627	414	N/A
	pass-Q	166	414	424
10%	pass-KV	631	1608	N/A
	pass-Q	544	1608	1023

Table 5 TTFT / TTIT (in ms) comparisons between TP8 and CP2 with different context lengths at batch size 1.

Context length	TP8		CP2+TP8	
	TTFT	TTIT	TTFT	TTIT
8K	1740	44.51	999	65.61
32K	7658	44.64	4015	65.66
128K	42010	46.26	21042	66.63

each iteration of the ring algorithm loop, which is repeated $N - 1$ times. The *All2All* time refers to the communication required in the merge attention step at the end of pass-Q algorithm. Note that for $T = 3200$, the sum of exposed pass-KV communication ($(N - 1)\cdot$ (*SendRecv* - Attn)) is longer than pass-Q *All2All*, resulting in better performance for pass-Q compared to pass-KV.

We further validate the analytical model in Algorithm 1 for predicting the selection of pass-KV vs. pass-Q from Table 3. When the KV cache miss rate exceeds 12.5% ($2 \cdot \frac{N_{KV}}{N_H}$ in Equation 1), pass-KV is always selected, meeting the 2nd condition in Algorithm 1. At 10% KV cache miss rate, pass-KV remains the choice since the number of new tokens T is sufficiently large, satisfying Equation 2 (with *SendRecv* hidden under Attn in Table 4); Around 5% cache miss rate (e.g., $T = 6400$), the differences between pass-KV and pass-Q is less than 1%, allowing for either option to be selected. When cache miss rate falls below 3.25%, pass-KV communication becomes exposed, leading to the selection of pass-Q. Specifically, at a 2.5% cache miss rate, the sum of the exposed communication in pass-KV ring loop is less than *All2All* exposed in pass-Q, resulting in the selection of pass-Q .

4.3 Decode Performance

Inference decode generates one output token at a time, resulting in a small amount of computation workloads and communication traffic. To avoid host kernel launch bottlenecks for these small kernels, we run both CP and TP decode with CUDA Graphs (Nvidia Blog, 2019).

Context Length Scalability: We benchmarked CP decoding performance with 2 nodes on GTT (using ring pass-Q decode algorithm in Section 3.5), and compare with TP8 decoding performance on 1 node using a single batch decode with various context lengths. As shown in Table 5, the TTIT of both TP8 and CP2 does not increase too much: For both TP8 and CP2, the computation and communication for linear layers stay the same while the latency of attention kernels increases with a longer context length.

Parallelism Scalability: We benchmarked different parallelization configurations up to four CP nodes to observe the scalability of both TP and CP. Table 6 shows that TTIT tends to be longer for both scaling TP and scaling CP. TTIT for scaling TP increases to 47 ms while TTIT for scaling CP increases to 71 ms. Both TP and CP have poor scalability for decoding because when adding more hosts. For TP, lower computation latency on linear layers is offset by increased communication latency increased.

For CP, as we increase the number of hosts, the effective length seen by each attention kernel decreases, so each individual attention op becomes faster (Table 7). However TTIT still degrates compared to CP=1, and the reason for that is two-fold: 1) Current implementation pads the number of queries to make it divisible by the number of ranks, which for B=1 means the total number of processed queries increases with CP. 2) The communication latency - sending Q chunks to the next rank at each iteration of the loop and all2all-exchanging partial attention outputs after the loop - also grows with the number of hosts. As a result, the total pass-Q attention latency and TTIT increase with CP.

In summary, context parallel is best suited for improving prefill performance and can be best leveraged with a

Table 6 TTFT / TTIT (in ms) comparisons between TP8, CP2, TP16, CP4, TP32 with 128K context length at batch size 1.

	TTFT	TTIT
CP1+TP8	42010	46.26
CP2+TP8	21042	60.23
TP16	29917	39.52
CP4+TP8	10950	71.31
TP32	19841	47.3

Table 7 Attention scaling with the number of CP hosts (Time in μs).

	Context length 128K, batch size 1		
	TP8	CP2+TP8	CP4+TP8
Effective context length	128K	64K	32K
Individual attention op	38.9	22.0	14.7
Attn (whole ring loop)	38.9	43.2	60.8
SendRecv	0	32.3	105.7
All2All	0	81.1	79.9
Whole pass-Q	38.9	157.7	238.6
	Context length 32K, batch size 4		
Effective context length	32K	16K	8K
Individual attention op	60.1	13.9	9.6
Attn (whole ring loop)	60.1	24.5	41.3
SendRecv	0	33.3	104.9
All2All	0	66.8	72.2
Whole pass-Q	60.1	136.6	180.6

serving system that decouples the parallelization scheme for prefill and decode (Qin et al., 2024; Zhong et al., 2024). For standalone deployment where prefill and decode are both on the same set of hosts, CP drastically improves the prefill latency, at the expense of decode latency regression (Removing batch padding and better overlap of computation and communication can help to minimize this regression).

5 Conclusion

In conclusion, our work highlights the effectiveness of context parallelism and ring attention variants in improving the efficiency of LLM inference for long-context scenarios. By leveraging up to 128 GPUs, we achieved near-linear scaling and significantly reduced latency, completing tasks with impressive speed and efficiency. Our implementation of the lossless exact pass-KV and pass-Q ring attention variants has been critical in supporting various full prefill, partial prefill, and decoding scenarios. The runtime heuristic adaptively selects pass-KV or pass-Q based on KV cache hit rate, optimizing their application for the most suitable scenarios.

As we keep improving LLM's capacity to understand increasingly longer and more complex context, one can expect diminishing utility with exact attention over all historical tokens. More efficient algorithms for retrieving a small subset of information from a much larger context to answer simple probe questions will be increasingly important. While context parallel is an efficient exact algorithm for scaling exact attention with more capacity, combining its processing power with an approximate retrieval algorithm for ultra-long context may be the best way to bound the processing latency for context window growth at and beyond 1M.

6 Acknowledgments

We express our gratitude to our outstanding colleagues for their significant contributions to various aspects of LLM inference. Special thanks go to Geonhwa Jeong, Jaewon Lee, Jason Park, Vlad Mihailescu, Zheng Yan, and Daniel Haziza for their invaluable efforts related to this paper. Our thanks also go to Chunqiang Tang for his early feedback and proofreading of the draft. Furthermore, we appreciate the leadership and support provided by Chunqiang Tang, Maxim Naumov, Amit Nagpal, Tony Liu, Changkyu Kim, Anca Agape, and Stephen Chen.

References

Claude with 200K context length. https://support.anthropic.com/en/articles/8606395-how-large-is-the-anthropic-api-s-context-window. Accessed: 2024-10-30.

Flash-decoding for long-context inference. https://pytorch.org/blog/flash-decoding/. Accessed: 2024-10-30.

Google's Gemini 1.5 Pro with 1M context length. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024. Accessed: 2024-10-30.

Nvidia GB200 NVL72. https://www.nvidia.com/en-us/data-center/gb200-nvl72/. Accessed: 2024-10-30.

GPT-4o with 128K context length. https://platform.openai.com/docs/models. Accessed: 2024-10-30.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

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

Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Minsik Cho, Mohammad Rastegari, and Devang Naik. Kv-runahead: Scalable causal llm inference by parallel key-value cache generation. *arXiv preprint arXiv:2405.05329*, 2024.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022. https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Gemini Team. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

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

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *arXiv preprint arXiv:2407.02490*, 2024.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

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

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

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

Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532*, 2024.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021.

Llama Team. The Llama 3 herd of models. 2024. https://arxiv.org/abs/2407.21783.

Meta Engineering. OCP summit 2022: Open hardware for ai infrastructure. https://engineering.fb.com/2022/10/18/open-source/ocp-summit-2022-grand-teton/, 2022. Accessed: 2024-10-30.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

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

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

Nvidia Blog. Getting started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019. Accessed: 2024-10-30.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *URL https://arxiv. org/abs/2407.00079*, 2024.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

Wenhan Xiong, Barlas Oğuz, Anchit Gupta, Xilun Chen, Diana Liskovich, Omer Levy, Wen-tau Yih, and Yashar Mehdad. Simple local attentions remain competitive for long-context tasks. *arXiv preprint arXiv:2112.07210*, 2021.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

Table 8 Notation table.

Notation	Description
N_H	Number of query heads (or attention heads)
N_{KV}	Number of key/value heads
D_H	Head dimension in Transformer
D	Model dimension in Transformer: $D_H \cdot N_H$
Q, K, V	Query, Key, Value tensors
B	Batch size (# of input sequences)
W	Model parameter size
T	Input sequence length
P	Previously cached KV length
e	Element data type size
C	Peak compute
BW	Peak comm bandwidth
bid	Decode batch id
$T_j^i \| P_j^i$	# of {new tokens | cached KV tokens} in i-th sequence sharded to CP rank j
L^i	KV length in i-th sequence: $max_{j=0}^{N-1}(P_j^i + T_j^i)$
N	Number of hosts/nodes/CP ranks
N_{TP}	TP group size (# of GPUs in a TP group)
$Q_k^s \| KV_k^s \| bid_k^s$	{Query | key and value tensors | batch id } on rank k, originally allocated to rank s
O_k^s	Attention output from Q_k and KV^s
$TP8, TP16$	Tensor parallel sharding over 8 GPUs on one node or 16 GPUs on two nodes
CP_N	Context parallel sharding on N nodes with TP8 for each node (same as $CP_N + TP8$)
$TTFT$	Time-to-first-token: latency for prefilling the whole input tokens
$TTIT$	Time-to-incremental-token: latency for decoding each output token

Table 9 Llama3 405B model configurations.

Parameter	Value
Layers ($\#layers$)	126
Model Dimension (D)	16,384
FFN Dimension	53,248
Attention Heads (N_H)	128
Key/Value Heads (N_{KV})	8
Parameter Size (W)	405 B

A Notations

We attach the notations used in this paper for reference in Table 8.

B MFU Calculation for 1M context length

We calculate the effective Model FLOPS utilization (MFU) (Chowdhery et al., 2023) in this section. The Llama3 405B model configurations are listed in Table 9. The total FLOPS are dominant by GEMM and Attention parts:

$$\text{Total FLOPS} = \text{GEMM FLOPS} + \text{ATTN FLOPS}.$$

- For GEMM, an W-parameter Transformer model requires $2 \cdot W$ matrix multiplication FLOPs for each token during inference:

$$\text{GEMM FLOPS} = 2 \times W \times T \times B.$$

- For Attention, the FLOPS is quadratic with respect to the context length T:

$$\text{ATTN FLOPS} = 1/2 \times 4 \times B \times T^2 \times D \times \#layers,$$

where 1/2 is from the causal mask, 4 is from 2 batch matmul and 2 FLOPS for multiplication and addition.

With input sequence length $T = 1M$, batch size $B = 1$, the parameter size $W = 405B$, we can get $\texttt{GEMM FLOPS} = 2 \times 405B \times 1M = 8.1 \times 10^{17}$. With the model dimension $D = 16384$, and number of layers $\#layers = 126$, we can derive $\texttt{ATTN FLOPS} = 1/2 \times 1M^2 \times 16384 \times 126 = 4.1 \times 10^{18}$. Attention FLOPS is more dominant compared to GEMM FLOPS. The total FLOPS is 4.9×10^{18}. With 77 seconds for 1M context length using 128 H100 GPUs, each H100 achieves $4.9 \times 10^{18}/77/128 = 502$ TF/sec. Note that with the standalone Flash Attention v3 causal attention benchmark using 8K context length on a single H100 (1M context length sharded across 128 H100 GPUs), we achieve 540 TF/sec. One caveat for the evaluation is that GTT/GTI (Section 4.1) are configured with power limited H100 GPUs (500 Watt) with lower memory bandwidth (96 GB HBM2e with 2.4 TB/sec instead of 80 GB HBM3 with 3.35 TB/sec), where the BF16 peak for each H100 is 800 TF/sec, instead of 989 TF/sec for H100 HBM3 with 700 Watt.

C Merge Attention

The idea of merging attention outputs from different keys/values originates from Online Softmax (Milakov and Gimelshein, 2018). Later this idea was reused in Flash Attention (Dao et al., 2022; Dao, 2023). Here we derive the equation to merge the partial attention outputs from different CP ranks.

The scaled dot production attention operates on query/key/value tensors $Q/K/V$. For simplicity, we don't consider various mask like causal masks (no batching or multiple attention heads either). There is one $Q/K/V$ corresponding to each sequence position. $Q/K/V$ at a given sequence position is a vector in the embedding space. The attention output is defined as

$$O = \texttt{Attn}(Q, K, V) = \texttt{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V,$$

where `softmax` is applied row-wise.

Assuming the size of row is R,

$$O = \sum_{i=0}^{R-1} V_i \cdot e^{Q \cdot K_i^T/\sqrt{d}} e^{-LSE},$$

where log-sum-exp LSE is defined as:

$$LSE = \log \sum_{i=0}^{R-1} e^{Q \cdot K_i^T/\sqrt{d}}.$$

In Section 3.4.2, we calculate the attention output and LSE on each CP rank k:

$$LSE_k^s, O_k^s = \text{Attn}(Q_k, KV^s),$$

with $s = 0, 1, ..., N-1$ on CP rank k.

Similar to blocked `softmax` computation in Flash Attention (Dao et al., 2022; Dao, 2023) and the derivation process in (Juravsky et al., 2024), we can get

$$O_k = \frac{\sum_{s=0}^{N-1} O_k^s \times e^{LSE_k^s - LSE_k^{max}}}{\sum_i^{N-1} e^{LSE_k^s - LSE_k^{max}}}, \tag{4}$$

where $LSE_k^{max} = \max_{s=0}^{N-1} LSE_k^s$.

In this way[5], we can combine attention output computed on different chunks of K/V for the same query to get attention on the whole K/V.

[5]Merge attention implementation is open sourced at https://facebookresearch.github.io/xformers/components/ops.html#module-xformers.ops.fmha.

D Analytical Model Selection Considering All2All

pass-Q merge attention requires an *All2All* (Section 3.4.3), whereas in pass-KV merge attention only needs to merge the partial attn results on local node (Section 3.4.2). When pass-KV communication is exposed, we want to compare the total of exposed pass-KV's communication time to the pass-Q's *all2all*, which is the time to send partial attention output and partial attention softmax log-sum-exp (LSE) (Appendix C):

$$Latency(All2All) = (N-1) \cdot \frac{(D+1) \cdot T \cdot e}{BW}$$

This means pass-Q has better prefill latency only if:

$$(N-1) \cdot \left(\frac{2(T+P)D \cdot e \cdot \frac{N_{KV}}{N_H}}{BW} - \frac{4 \cdot T \cdot D \cdot (T+P)}{N \cdot C} \right)$$

$$\geq (N-1) \cdot \frac{(D+1) \cdot T \cdot e}{BW}$$

Assuming $D \approx D+1$, through algebraic rearrangement, we get:

$$2 \cdot \frac{N_{KV}}{N_H} - \frac{4T \cdot BW}{N \cdot C \cdot e} \geq \frac{T}{T+P} \tag{5}$$

Compared to (1), this shows that considering *All2All* decreases the KV cache miss rate threshold for selecting pass-Q .

Algorithm 5 is the adjusted heuristic algorithm to select between pass-KV and pass-Q , considering *All2All* used in merge attention in pass-Q .

Algorithm 5 Pass-KV vs. Pass-Q Partial Prefill Heuristics

```
if $T \geq N\frac{C \cdot N_{KV} \cdot e}{2 \cdot N_H \cdot BW}$ or $\frac{T}{T+P} \geq 2 \cdot \frac{N_{KV}}{N_H} - \frac{4T \cdot BW}{N \cdot C \cdot e}$ then
    pass-KV
else
    pass-Q
end if
```

E Heuristic based on empirical data

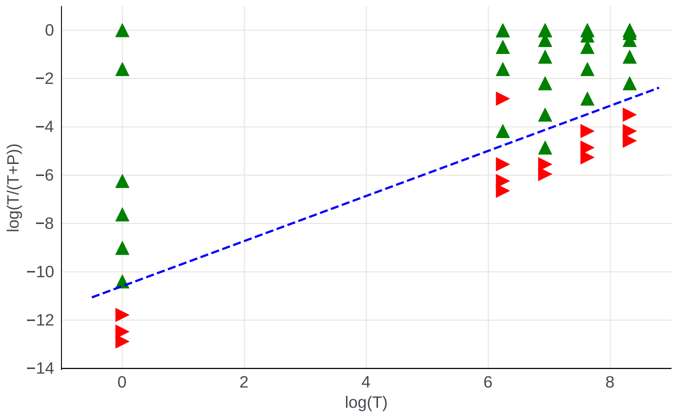


Figure 10 A heuristic model using empirical data points. Green: prefer pass-KV , Red: prefer pass-Q

For practical uses, we further establish a simplified heuristic to choose between `pass-KV` and `pass-Q` based on emprical data points. Particularly we collected data points for various combinations of T and $T/(T+P)$, and establish an empirical formula:

$$h(T, P) = \alpha \cdot \log(T) + \beta \cdot \log\left(\frac{T}{T+P}\right) + \gamma$$

We prefer `pass-KV` when h evaluates to a positive value and prefer `pass-Q` otherwise. We fit empirical data points to this formula with parameters: $\alpha = -1.059$, $\beta = 1.145$ and $\gamma = 12.112$, as show in Figure 10. One way to interpret the heuristic is that, for each particular T, there is a threshold for $T/(T+P)$ based on which we should switch from `pass-Q` to `pass-KV` for best performances, and the threshold increases as T increases.

Note that we do not expect the linear model to perfectly capture all cases, so some misclassifications are present due to variances and other factors, but the general trend is obvious. We inspected the misclassified data points, and they turned out to be the ones where the differences between the two strategies were relatively small ($< 1\%$). In practice we can run this heuristic at the beginning of each round and get the best of both worlds.