

Mesh-Attention: A New Communication-Efficient Distributed Attention with Improved Data Locality

Sirui Chen^{‡*} Jingji Chen^{§*} Siqi Zhu^{¶††} Ziheng Jiang[§] Yanghua Peng^{§***} Xuehai Qian^{†***}

[†]Tsinghua University [‡]Purdue University [¶]University of Illinois Urbana-Champaign [§]ByteDance Seed

Abstract

Distributed attention is a fundamental problem for scaling context window for Large Language Models (LLMs). The state-of-the-art method, Ring-Attention, suffers from scalability limitations due to its excessive communication traffic. This paper proposes a new distributed attention algorithm, *Mesh-Attention*, by rethinking the design space of distributed attention with a new matrix-based model. Our method assigns a two-dimensional tile—rather than one-dimensional row or column—of computation blocks to each GPU to achieve higher efficiency through lower communication-computation (CommCom) ratio. The general approach covers Ring-Attention as a special case, and allows the tuning of CommCom ratio with different tile shapes. Importantly, we propose a greedy algorithm that can efficiently search the scheduling space within the tile with restrictions that ensure efficient communication among GPUs. The theoretical analysis shows that Mesh-Attention leads to a much lower communication complexity and exhibits good scalability comparing to other current algorithms.

Our extensive experiment results show that Mesh-Attention can achieve up to $3.4\times$ speedup ($2.9\times$ on average) and reduce the communication volume by up to 85.4% (79.0% on average) on 256 GPUs. Our scalability results further demonstrate that Mesh-Attention sustains superior performance as the system scales, substantially reducing overhead in large-scale deployments. The results convincingly confirm the advantage of Mesh-Attention.

1 Introduction

Large language models (LLMs) [26] show impressive capabilities in completing various real-world tasks such as AI agents [15], document summarization [25], virtual assistants [19], video understanding [5, 24], and code comple-

tion [21]. For LLMs, it is important to scale to larger context window (i.e., the maximum number of tokens that can be processed by the model) so that greater amount of information (e.g., longer documents or videos) can be leveraged when handling tasks. For this reason, the size of context window is a crucial metric when comparing different LLMs, and supporting longer context window has become an appealing selling point. For example, Gemini 2.5 Pro [9] supports a context length of 1 million tokens while Llama 4 Scout [17] can support up to 10 million tokens.

Scaling context window size is challenging because the computation and memory requirements of attention—the core component of LLMs—increase drastically with the increasing context window size. For this reason, *distributed attention* is intensively researched to ensure that long context window can leverage increasing amount of computation and memory resource. Efficient distributed execution of attention operation turns out to be a delicate problem that requires the joint consideration of parallelism and effective overlap of computation and communication, which can be hardware platform dependent. The recently proposed schemes suffer from different drawbacks. Ulysses [13] cannot support more GPUs than the number of attention heads since it leverages head parallelism. The widely used Ring-Attention [14] overlaps communication and computation nicely by dividing the whole procedure into multiple steps equal to the number of GPUs. However, it suffers from excessive communication.

Figure 1 (a) illustrates an example of Ring-Attention with 9 GPUs, where Q and KV tensors are partitioned along the *sequence* dimension into 9 chunks and distributed among the GPUs. Conceptually, attention operation needs to perform the computation between *each pair of Q and KV*. For a GPU, the computation of its local Q-KV pairs, i.e., both Q and KV value are local, can be performed without communication with other GPUs; while the computation between its local Q (or KV) and remote KV (or Q) involves communication which should be overlapped with computation as much as possible through delicate scheduling. In Ring-Attention, the design principle is for each GPU to keep its local Q (or KV) and let the remote

*Equal contribution.

††Work done during internship at ByteDance Seed.

**Corresponding authors.

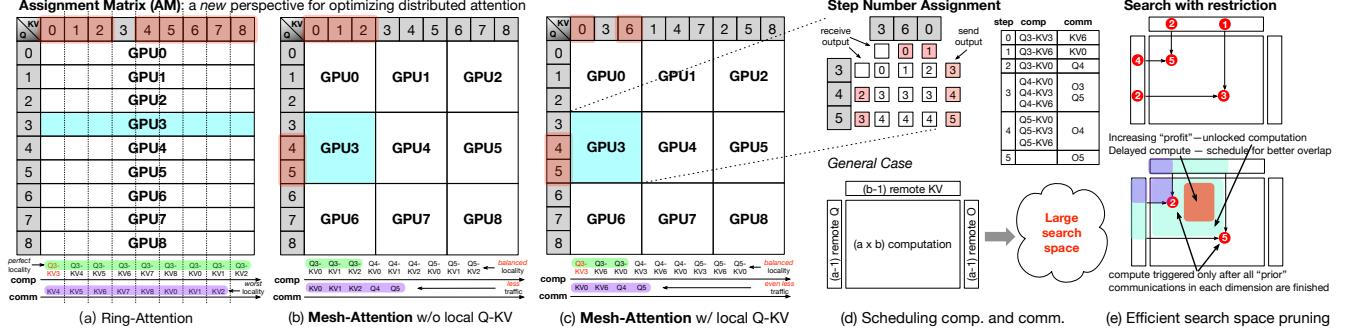


Figure 1: From Ring-Attention to *Mesh-Attention*

KV (or Q) pass through a *logical ring* among all GPUs. For each GPU, the whole operation is divided into 9 steps where the computation of each step and the communication initiated at the end of previous step are overlapped. At the end of the 9th step, the partial results produced by all completed steps in each GPU are reduced along the Q (or KV) dimension, and each GPU obtains part of the output chunk.

From the design of Ring-Attention, we can see that the key decision to make is which GPU should perform certain computation and when the communication should be scheduled. We can specify such decisions through an *assignment matrix (AM)* where vertical and horizontal direction correspond to the chunks of Q and KV, respectively, and each matrix element $AM[i][j]$ specifies the GPU that is responsible for performing the computation between Q_i and KV_j . The communication is implicitly expressed based on the requirement that before a certain computation is performed, its input data, if residing in a remote GPU, should be communicated to the corresponding GPU. Through this abstract model, Ring-Attention among 9 GPUs corresponds to the AM in Figure 1 (a), where each GPU performs the computation on its local Q chunk and all KV chunks (one local and 8 remote chunks). Since each KV chunk has two units (K and V), while each Q chunk has one unit, for Ring-Attention, the total amount of communication for each GPU is $2 \times 8 = 16$ units.

The matrix-based model provides new insights to analyzing the efficiency of a distributed attention implementation. Specifically, we can calculate the *communication-computation (CommCom) ratio*, which quantifies how much communication is required to perform certain amount of computation. For Ring-Attention, CommCom ratio is $(8 \times 2)/9 = 16/9$ ¹. CommCom ratio resembles the familiar concept of *locality* that is not intuitively revealed without the matrix-based model: locality quantifies how much remote chunks are communicated for Q or KV. For Ring-Attention, Q achieves *perfect locality* since there is no communication for Q, while

¹One may wonder why not using “col-wise” assignment and communicate the smaller Q, it is because in the end, each GPU needs to obtain the results of its local Q with all remote KV. Communicating Q incurs smaller amount of communication for the inputs but extra communication to distribute the outputs. There is indeed recent work exploiting the hybrid data-size dependent schedule, see [23].

KV exhibits *worst locality* since all remote KV chunks are communicated. As a general observation in system research, any “extreme” policy in the design space is rarely the optimal choice. Ring-Attention’s row-wise assignment leads to excessive communication that scales linearly with the sequence length, e.g., with a hidden size of 4096 and a sequence length of 1M tokens, the total size of the KV tensor is 8GB, which must be fully received by each GPU. Experimentally, the communication can take 91.5% of the forward time with 128 GPUs and a sequence length of one million tokens, showing significant overhead. It explains the importance of optimizing the computation and communication schedule of distributed attention.

This paper aims to rethink the design space of distributed attention through the new matrix-based model, and proposes *Mesh-Attention*, which assigns GPUs to *tiles* in assignment matrix (AM), instead of row-wise (communicate KV) or col-wise (communicate Q). This approach allows the fine-tuning of locality in *both dimension* to achieve the optimal implementation. Figure 1 (b) shows one possible way to assign computation to each GPU in Mesh-Attention with (3×3) -tile. Note that Mesh-Attention does not require square tile, but assumes the equal amount of computation and the same shape of tile for each GPU. In this setting, each GPU in the diagonal (i.e., GPU 0/4/8) communicates $2 \times 2 + 2 = 6$ units of data, while each other GPU communicates $3 \times 2 + 2 = 8$ units of data. However, extra communication is needed to distribute the output: each GPU needs to communicate 1 unit of data (the result of Q-KV computation) to two other GPUs in the same row, e.g., GPU0 communicates with GPU1 and GPU2. Putting all together, the total communication amount is $6 \times 3 + 8 \times 6 + 9 \times 2 = 84$ units of data, a reduction from $16 \times 9 = 144$ for Ring-Attention.

In principle, the assignment of tile to each GPU should ensure that the GPU performs the local Q-KV computation, if possible. We name it as *local Q-KV property*. Ring-Attention satisfies the property trivially by not communicating Q at all, but the earlier assignment in Mesh-Attention only satisfies the property for the GPUs in the diagonal of AM. Fortunately, we can gain local Q-KV property by cleverly *rotating* the indices in KV (or Q) dimension. For our example of square (3×3) -

tile, we keep the indices in Q dimension and the position of each GPU fixed, and then rotate the indices in KV dimension such that the three indices for each GPU column are exactly the IDs of the GPUs in the column. We can verify the property in the revised AM shown in Figure 1 (c). For the general tile size with shape $a \times b$, where $a \neq b$, we can derive a similar procedure, to be discussed in Section 3.2. With this property, the total amount of communication is further reduced to $6 \times 9 + 9 \times 2 = 72$. In a nutshell, Mesh-Attention increases the CommCom ratio and ensures better GPU utilization. Our theoretical analysis in Section 3.8 shows that Mesh-Attention can reduce the asymptotic communication complexity by a factor of \sqrt{n} , where n is the number of GPUs.

Reducing the amount of communication is only half of the problem, the other half is how to efficiently schedule the execution to maximize the overlapping of computation and communication. Notably, the communication pattern of Mesh-Attention is more complex than Ring-Attention: not only transferring KV chunks but also Q chunks and partial outputs to aggregate output for each GPU. This brings new challenges when developing efficient communication and computation overlapping. To achieve that, the overall strategy is similar to Ring-Attention: divide the whole procedure into multiple steps, and schedule the communication and computation operations into each step, aiming to maximizing the overlapping. However, Ring-Attention’s row-wise AM leads to *only one* sensible schedule: during each step, each GPU performs one Q-KV computation, while the needed KV chunk for the next step is concurrently transferred. For Mesh-Attention, the scheduling of computation and communication within each tile for a given GPU constitutes a *considerable search space*.

To facilitate the understanding and the space search, we express each schedule as the assignment of a *step number* $s \in [0, k)$ to each element of the part of AM for each GPU, as well as the part of rows for Q chunk and columns for KV chunk. Here, k is the number of total steps. In this paper, we assume the schedule of all GPUs are the same, with the identical tile shape for all GPUs, we only need to consider one such assignment. Figure 1 (d) (top) shows one example of assignment for a (3×3) -tile with 5 steps. For a general $(a \times b)$ -tile shown in Figure 1 (d) (bottom), with simple calculation, we need to assign to $(a \times b)$ elements for computation, to $2 \times (a - 1) + (b - 1)$ elements for communication². Since each element has k choices, there are in total $k^{(2 \times a + b - 3 + a \times b)}$ possible assignments—indeed a large search space. Clearly, the step number of communication (for remote Q and KV) must be smaller than that of the computation dependent on the remote data. However, expressing such constraint in search process is also difficult.

To limit the search space, we introduce two *restrictions*:
(1) To perform the computation in $AM[i][j]$, all Q_{ii} and KV_{jj} ,

²With local Q-KV property, a GPU always needs to compute based on its local Q and KV chunk, which incur no communication. Similarly, the GPU does not send partial outputs to others for one of the rows in the tile.

where $ii < i$ and $jj < j$ must be received; and (2) each step contains at most one communication, i.e., there are at least $(2 \times a + b - 3)$ steps. Based on the restrictions, the scenario in Figure 1 (e) (top) cannot happen, as shown in Figure 1 (e) (bottom). We propose a greedy algorithm to identify the optimized schedule that aims to maximize the overlap by scheduling a proper number of computation in each step. The insight of the algorithm is: (1) the *profit* of choosing a communication is considered as ratio between the elements in the tile that become *ready-to-execute* (a.k.a. “unlock”) and the communication amount, the algorithm chooses the communication with higher profit; (2) place “just enough” computation into each step, i.e., some ready-to-execute computation can be delayed to later steps to ensure better overlap. Figure 1 (e) (bottom) shows that based on the first restriction, the profit of communication increases as the communication of Q and KV proceed, thus it is important to intentionally delay some unlocked computation and schedule them with later communication step.

We implemented a distributed attention library based on our proposed Mesh-Attention algorithm. Our extensive experiment results show that Mesh-Attention can achieve up to $3.4 \times$ speedup ($2.9 \times$ on average) and 85.4% (79.0% on average) communication reduction under a configuration of 256 GPUs comparing to Ring-Attention. The experimental results convincingly confirm the advantage of Mesh-Attention.

2 Background

2.1 LLMs and Attention

Large language models (LLMs) have emerged as a cornerstone of modern natural language processing, demonstrating remarkable fluency and versatility across tasks such as question answering, summarization, and dialogue generation [26]. At their core, most state-of-the-art LLMs are built upon the transformer architecture [22], illustrated in a simplified form in Figure 2a. Within each layer, the input embeddings first pass through a self-attention block and then through a position-wise feed-forward network (FFN) composed of a multilayer perceptron (MLP) block, with residual connections and layer normalization applied around both blocks. The attention mechanism [22] is the key component of each transformer layer, which computes pairwise affinities between tokens via learned query, key, and value projections.

In the attention layer, when the sequence length is large, the point-wise projection operations involving W_Q, W_K, W_V and W_O are not the primary source of overhead. In contrast, the projected query(Q), key(K), and value(V) activations constitute the major memory overhead when the sequence length is dominant, and the core attention operation performed on them is the main source of computational cost:

$$P = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)$$

$$O = PV$$

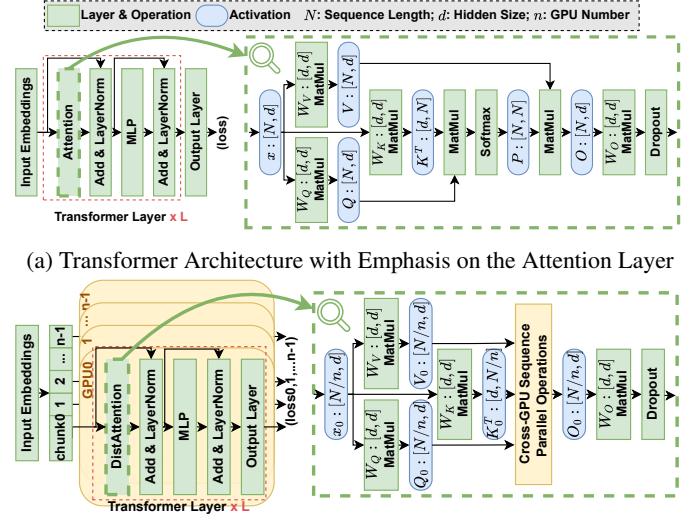
The matrix multiplication between Q and K^T computes a pairwise similarity score between every query-key pair, quantifying the relevance of each token to all others in the sequence. These scores are scaled by $1/\sqrt{d}$ and applied with softmax, producing a normalized probability matrix P , where each row represents the distribution of attention weights for a given query token over all key tokens. Finally, multiplying P by V aggregates the value vectors across the sequence according to the learned attention weights, yielding the output representation for each token. From the perspective of token-to-token relationships, the attention mechanism can be viewed as operating along two dimensions: each token's query interacts with the keys and values of all other tokens to compute its own output.

The optimization of this operation is one of the most crucial aspects in accelerating LLMs. One of the most widely used techniques is FlashAttention, which reorders and fuses the sequence of matrix multiplication and softmax to drastically reduce memory bandwidth and intermediate storage [6, 7]. More recently, there has been intense interest in extending LLMs to long contexts both for training on extended text corpora [1] and for inference over multi-page inputs [3]. However, scaling attention to thousands or even millions of tokens imposes a disproportionate burden on the model's memory and compute resources, particularly within the self-attention block due to the quadratic compute and memory complexity related to the sequence length.

To alleviate these bottlenecks, a variety of distributed and hierarchical attention schemes have been proposed. Beyond conventional parallel optimization methods such as data parallelism [10], tensor parallelism [20], and pipeline parallelism [11, 12], a new parallelization dimension applied along the sequence, i.e., *sequence parallelism*, has been proposed to address the challenges posed by long context tasks. In sequence parallelism shown in Figure 2b, the input embedding x is sharded sequence-wise to chunks and distributed to all GPUs. Following the projection step, each GPU holds only its local chunk of the Q, K, and V tensors. Consequently, cross-GPU communication is required to perform the core distributed attention operations in which all tokens are interacted along the Q-KV dimensions. Ultimately, each processor retains only the O chunk corresponding to its assigned input region. Next, we discuss several recent distributed attention schemes.

2.2 Ring-Attention

Ring-Attention [14] is a type of sequence parallelism that splits the Q, K and V tensors across all GPUs and performs



(b) Sequence parallelism. Different Methods Have Different Implementation for Cross-GPU Sequence Parallel Operations

Figure 2: Transformer Architecture

cyclic rotation to pass on KV (K and V) tensor chunks. Each GPU computes attention with the received KV tensor chunks on its local Q chunk, accumulating partial outputs until a complete result is assembled.

By leveraging the online softmax normalization trick [18], attention computation can be partitioned into multiple small blocks along both the Q and KV dimensions, where each blockwise attention computation yields a partial output that is incrementally summed into the final O (output) tensor. Ring-Attention applies this blockwise scheme to overlap communication with computation when aggregating the local O tensors of each GPU.

Referring again to Figure 1 (a), the Q and KV tensors are evenly split along the dimension of the sequence into 9 chunks $\{Q_i\}_{i=0}^8$ and $\{KV_i\}_{i=0}^8$ across 9 GPUs. Initially, GPU i holds chunks Q_i and KV_i . All 9 KV chunks are passed on through the logical ring until each GPU i has collected all of them, while performing blockwise attention with its local Q_i at each passing step. The blockwise attention computation can be described as:

$$O_{i,j}, lse_{i,j} = \text{Attention}(Q_i, KV_j)$$

where $O_{i,j}$ stands for the block output chunk with Q_i and KV_j , and $lse_{i,j}$ stands for the log-sum-exp (lse) coefficient chunk used for online softmax. When GPU i iterates over j to apply the above formula, where $j = 0, 1, \dots, 8$, all the output chunks and the lse chunks are cumulatively reduced into O_i and lse_i by online softmax. O_i will have the same dimensions as Q_i and is passed on to subsequent layers of the model. Since the lse chunk is a vector, its size is negligible compared to an O chunk. Therefore, in the following discussions, when we refer to an O chunk, we implicitly include the corresponding lse chunk without explicitly writing it out.

Although Ring-Attention overlaps the communication with

blockwise attention computation, the cyclic exchange of large KV chunks among all GPUs can become a communication bottleneck in low-bandwidth, large-scale deployments. In experiments on machines configured as 128 GPUs, we found that during the prefill stage of inference on inputs of length 1M, Ring-Attention spends 91.5% of its time waiting for communication without overlapping with computation.

2.3 Other Related Works

Yang et al. [23] made a slight modification to Ring-Attention by designing a mode that passes Q chunks instead of KV chunks along the ring, selecting the better option depending on the context. Passing Q chunks requires an additional reduce-scatter communication at the end to aggregate the partial results, but is otherwise similar to Ring-Attention.

The DeepSpeed-Ulysses [13] (DS-Ulysses) is another sequence parallelism strategy, orthogonal to Ring-Attention. Input embeddings are first sharded along the sequence dimension and distributed across all GPUs. During the attention layer, each GPU performs an initial all-to-all exchange that transposes chunks between the sequence and head dimensions, which ensures that every GPU receives Q and KV chunks containing one or more complete heads, permitting full head-wise attention computation locally. A second all-to-all exchange then reverses this transpose, swapping the head and sequence dimensions so that each GPU ends up with its assigned segment of the sequence-level output, which is then passed to the model’s subsequent layers.

Although efficient all-to-all implementations can dramatically reduce communication overhead, DeepSpeed-Ulysses’s scalability remains constrained: the maximum number of GPUs it can leverage is ultimately limited by the number of heads in the multi-head attention mechanism. To address this limitation, USP, or Unified Sequence Parallelism, [8] proposes a hybrid approach that fuses DeepSpeed-Ulysses and Ring-Attention, since they are fundamentally orthogonal sequence-parallelism schemes. Once DeepSpeed-Ulysses has exhausted its head capacity, additional scalability is provided by Ring-Attention.

Startrail [16] mitigates the communication bottleneck of Ring-Attention by introducing an additional parallel dimension. In this design, GPUs are first partitioned into disjoint groups, and at the outset of the attention computation, KV tensor shards are disseminated across all groups via an All-Gather collective. Its 3D topology reduces the communication complexity compared to Ring-Attention. However, the intricate grouping strategy and the ensuing inter-group collectives incur redundant data transfers, indicating scope for reducing the constant factors in communication complexity. Furthermore, the limited ability to overlap these collective communications with local computations constrains the overall performance benefits.

3 Mesh-Attention

3.1 Definition and Analysis

In this section, we define the concepts we explained in the Introduction in general form and highlight the drawbacks of Ring-Attention.

Assignment Matrix (AM): consider a distributed attention operation where Qs and KVs are partitioned into n chunks and KVs are partitioned into b chunks, the *assignment matrix (AM)* is a $n \times n$ matrix, the value of $AM[i][j]$ is the ID of the GPU that is responsible for computing Q-KV pair between $Q[i]$ and $KV[j]$.

Communication-Computation Ratio (CommCom): the ratio between communication and computation for the whole distributed attention among all GPUs, or the ratio for an individual GPU.

Local Q-KV Property: a GPU is responsible for computing the Q-KV pairs between its local Q and KV chunks.

Step Number Assignment: given k steps, and an $(a \times b)$ -tile for a given GPU, the assignment of a step number from $0 \dots (k - 1)$ to each element of AM for computation as well as the $(a + b)$ input Q and KV chunks and a output O chunks for communication. For the local input chunks, the step number is 0 since the data is readily available; for the local output chunks the GPU only receives partial output from other GPUs but does not send it.

Ring-Attention: it is a distributed attention algorithm where a GPU ID is assigned to one *row* of AM, reflecting the principle of a GPU always computing the Q-KV pairs between its local Q chunk and all local and remote KV chunks.

Ring-Attention suffers from *imbalanced locality*: achieving the perfect locality for Q without accessing any remote Q chunks, and the worst locality for KV accessing all remote KV chunks. It leads to excessive communication amount.

3.2 Tiling Based Workload Distribution

In an arbitrary setting of n GPUs, any factorization of $n = a \times b$ presents a valid tiling size that can be mapped onto the AM. We arrange GPU 0 through GPU $n - 1$ to the tiles in AM in a row-first manner. The Q chunk indices along the left column remain 0 to $n - 1$, while the KV chunk indices along the top row are modified as follows: for every consecutive group of b indices, each is replaced with the indices of the b GPUs located beneath them in the tiles, as shown in Figure 1. This concludes the workload assignment of Mesh-Attention.

Thanks to the reordering of KV indices on the AM, each GPU not only computes the local Q-KV attention block, but also shares symmetric properties, reflected in the following grouping strategy:

- **Q groups:** Each Q group i is formed by a group of a consecutive GPUs, indexed by $\{ai + x\}_{x=0}^{a-1}$. $i \in \{0, \dots, b - 1\}$

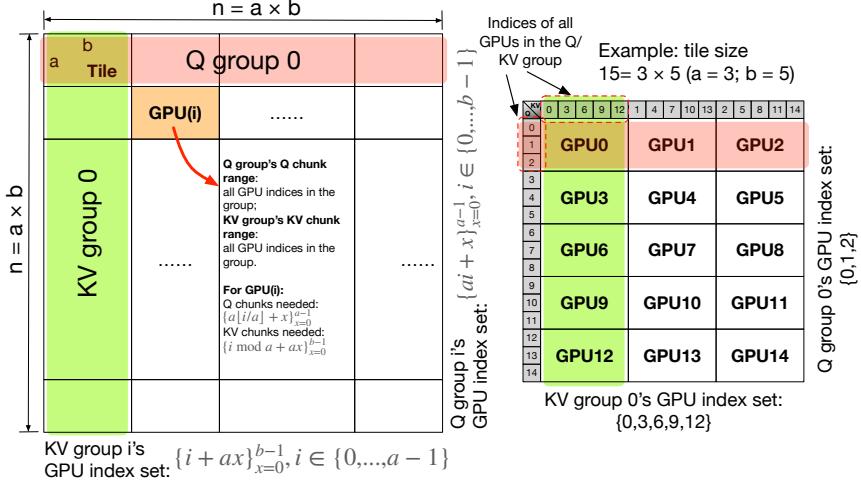


Figure 3: General Mesh-Attention

- **KV groups:** Each KV group i is formed by GPUs that share the same residue class modulo a , indexed by $\{i + ax\}_{x=0}^{b-1}$, $i \in \{0, \dots, a-1\}$

Each GPU i is a member of exactly one Q group and one KV group. Q chunks of indices $\{a[i/a] + x\}_{x=0}^{a-1}$ are acquired through the Q group; KV chunks of indices $\{i \bmod a + ax\}_{x=0}^{b-1}$ are acquired through the KV group. Because Q_i and KV_i are held by GPU i initially, the communication volume is $a - 1$ Q chunks, plus $b - 1$ KV chunks, plus $a - 1$ O chunks.

If we connect the GPUs within each group in a ring and let all communications occur along these rings, the entire system forms a wrap-around mesh. So far, we observe an interesting property: all GPUs can perform communication and computation in synchrony: each GPU needs to know only the predecessor and successor in its Q or KV group to perform identical, lock-step communication operations, thereby eliminating bubble latency caused by imbalance.

3.3 Computation Flow

Algorithm 1: Computation Flow of Forward Pass of Mesh-Attention

- 1 All-gather Q chunks on every Q group
 - 2 All-gather KV chunks on every KV group
 - 3 **Each GPU i :** **for** each Q-KV pair (Q_u, KV_v) in the gathered Q chunk slots and KV chunk slots **do** Compute attention (Q_u, KV_v) and update local $O_u^{(i)}$ using online softmax
 - 4 Reduce-scatter O chunks within every Q group, using online softmax as the reduce operator
-

Based on the tile-based workload distribution, we specify the computation flow of each GPU. The partition in Figure 3 implies that the work among all GPUs are *symmetric*, i.e., the computation flow of each is identical, thus we only need to specify that once.

The computation flow for each GPU in Mesh-Attention is specified in Algorithm 1. Each GPU must acquire remote chunks within the designated Q and KV group before performing computation using the remote and local chunks. After all computations are finished, a corresponding O chunks are produced, of which $a - 1$ must be sent to the other $a - 1$ GPUs in the Q group that “own” the corresponding local Q chunks. Overall, after all O chunks are produced, they must be reshuffled: each GPU sends its $a - 1$ non-local chunks to appropriate peers and receives the partial outputs of its own chunk back from them.

Algorithm 1 describes the functional computation flow without any overlapping and optimization. The remote Q and KV chunks are acquired through two all-gather operations, after the attention computation, the output reshuffling and reduce are implemented by a reduce-scatter operation. Specifically, after line 1 and line 2, any GPU i acquired Q chunks $\{a[i/a] + x\}_{x=0}^{a-1}$ and KV chunks $\{i \bmod a + ax\}_{x=0}^{b-1}$, which are determined by the workload assigned in the AM. Therefore, in line 3, GPU i is able to compute all the Q-KV blocks assigned to it in AM. In line 4, while GPU i sends out its $O^{(i)}$ chunks indexed $(\{a[i/a] + x\}_{x=0}^{a-1} - \{i\})$, it simultaneously receives the corresponding $O^{(\dots)}$ chunks i from the other $(a - 1)$ GPUs in the same Q group and reduces them into the final output O_i .

Challenges of Obtaining Efficient Scheduling. An efficient distributed attention implementation requires delicate schedule of computation and communication so that the two can be overlapped as much as possible. The above computation flow poses three problems. First, the algorithm just specifies the operations but it does not provide an intuition on how to efficiently overlap communication and computation. Second, as outlined in the Introduction, the search space is huge when we phrase the problem as placing computation operations into each communication slot. With a large number of GPUs, it leads to a fundamental difficulty. Third, the

latency of each communication slot is not fixed, which depends on both hardware platform and source/destination of data to be transferred. It prevents the accurate estimation of the number of computation operations to be placed in a slot.

3.4 Intra-Tile Communication Scheduling

To tackle these challenges, we leverage a key observation in Mesh-Attention’s communication pattern: while the computations in AM are partitioned by two-dimensional tiles among GPUs, each Q/KV group we defined earlier is one-dimensional. Thus, we can apply the well-known computation and communication overlapping techniques in Ring-Attention within each Q/KV group, where the GPUs inside the group form a logical ring. This idea also solves the latter two problems at the same time: (1) more restricted communication and computation schedule within Q/KV groups shrinks the search space due to the additional assumption; and (2) decomposing the all-gather and reduce-scatter operations leads to small communication steps with fixed latency, enabling more accurate selection of computation operations to be overlapped.

Specifically, the collective communication within a Q/KV group can be decomposed into fine-grained point-to-point (P2P) communication steps: In a step of all-gather, each GPU sends its local data to its successor and forwards received data from its predecessor, continuing until all GPUs have collected the full set of data. In a step of reduce-scatter, each GPU sends one chunk to its successor and reduces received chunks with the appropriate local item, continuing until each GPU obtains the final reduced result for its assigned chunk.

The fine-grained point-to-point communication steps within a Q/KV group are performed synchronously among GPUs in the group with different sources and destinations of transferred messages. To derive a concise and identical specification of operations for all GPUs, we introduce a *mapping* shown in Table 1 from local chunk index, denoted as $Q\#u$ or $KV\#u$ or $O\#u$, to the global chunk index based on the computation distribution in Figure 3. We assume a tile shape of $a \times b$ and a total of $n = a \times b$ tiles. The mapping is given in Table 1, which specifies the global chunk index as a function of GPU index i , a , b and n . Each GPU has its own output chunk $O\#u$ (note the superscript i), because the partial output produced by each GPU is different depending on its position in the logical ring within the group.

Table 1: Local to Global Chunk Index Mapping on GPU_i

Chunk Map	Global Chunk Index
$Q\#u \rightarrow Q_v$	$v = a\lfloor i/a \rfloor + (i+u) \bmod a$
$O\#u \rightarrow O_v^{(i)}$	$v = a\lfloor i/a \rfloor + (i+u) \bmod a$
$KV\#u \rightarrow KV_v$	$v = (i+au) \bmod n$

Following the decomposition method and insights, we define the P2P operations for each GPU below. The successor

and predecessor are well-defined for each GPU based on the AM derived in the procedure described in Section 3.2.

- **Three zero-initialized variables** i_q , i_{kv} , and i_o are introduced to track the progression of P2P operations. Initially, each GPU holds its local Q and KV chunks, notated as $Q\#0$ and $KV\#0$ respectively.

- **Recv Q/KV:** 1) send $Q\#i_q/KV\#i_{kv}$ to the successor in the Q/KV group, 2) receive $Q\#(i_q+1)/KV\#(i_{kv}+1)$ from the predecessor, 3) increase i_q/i_{kv} by 1.

- **Send O:** 1) send $O\#(i_o+1)$ to the successor in the Q group, 2) receive $O\#((i_o+2) \bmod a)$ from the predecessor and reduce into the local one using online softmax, 3) increase i_o by 1.

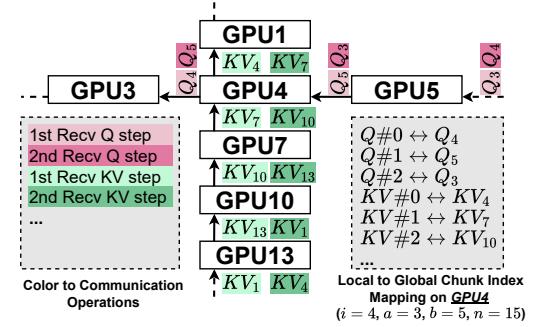


Figure 4: Two Steps of GPU 4’s Recv Q/KV Operations Based on the 15-GPU Example in Figure 3

It is important to note that all GPUs in the group would execute Recv Q, Recv KV, or Send O synchronously, and except the first step, the GPUs send the Q/KV chunk received in the previous step. For better understanding, Figure 4 illustrates two steps of GPU 4’s operations in Q and KV groups based on the AM in Figure 3. In general, all-gather on Q chunks is decomposed to $a - 1$ Recv Q operations on every GPU in the Q group; and all-gather on KV chunks is decomposed to $b - 1$ Recv KV operations on every GPU in the KV group. The reduce-scatter of O chunks is decomposed to $a - 1$ Send O operations on every GPU in the Q group.

The **Recv Q/KV** and **Send O** defined earlier are called by each GPU in the Q/KV group without explicitly specifying the data to be send. Instead, the transferred data is implicitly determined by the mapping in Table 1 and the number of times that certain operation is called. This restricts the schedule of computation and reduce search space. In general, the computation on $Q\#i-KV\#j$ becomes *ready-to-execute* after the assigned GPU receives $Q\#i$ and $KV\#j$, but with ring-based communication within Q/KV group, $Q\#i-KV\#j$ becomes ready-to-execute *after* $i - 1$ *Recv Q operations* and $j - 1$ *Recv KV operations* have been performed in prior steps. Essentially, within a logical ring, the Q and KV chunks are received *in certain order*, thus, the computations become ready-to-execute also in certain order, the arbitrary reordering shown in Figure 1 (e) upper part is naturally disallowed—reducing search space.

Moreover, the i -th **Send O** (starting from 1-st since each GPU only sends partially reduced output for remote rows in AM) can be only performed after attention computations for $\{Q\#(i)-KV\#(j)\}_{j=0}^{b-1}$ have been performed in prior steps.

3.5 Automatic Scheduling Generation

Based on ring-based decomposed communication within Q/KV group, we propose a greedy algorithm to derive efficient schedule that aims to maximize the overlapping of computation and communication. Referring to Figure 1 (d), the problem can be framed as assigning a step number to each element in AM of a tile and the dependent communication of Q/KV chunk. We follow three principles in choosing step number assignment.

- **Making compute ready-to-execute fast and balanced.**

The transfer of each remote Q/KV chunk would convert certain computation blocks to ready-to-execute, without careful schedule, this conversion can be *too slow*, e.g., Ring-Attention shown in Figure 5 (a), each communication triggers exactly one computation block, directly hindering performance; or *imbalanced*, e.g., Mesh-Attention with row-first schedule shown in Figure 5 (b), the first two communications each triggers one computation block while the last two each triggers three computation blocks, leading to sub-optimal amount (either insufficient or excessive) of computations to be overlapped with communication.

To solve the problem, statically, we profile c_Q , c_{KV} , and c_O , which represent the least number computation blocks to fully hide the time of transferring one Q, KV, or O chunk, respectively; dynamically, we obtain n_Q and n_{KV} , which represent the number of computation blocks that are made ready-to-execute due to a Recv Q/KV. Thus, the *profit* of the chosen Q or KV communication can be considered as $\frac{n_Q}{c_Q}$ or $\frac{n_{KV}}{c_{KV}}$. In each step, with the goal of obtaining higher profit, we can compare $\frac{n_Q}{c_Q}$ and $\frac{n_{KV}}{c_{KV}}$, and choose Q or KV communication that leads to higher value.

- **Triggering just enough computation for overlapping.**

Even with good choices of Q and KV, it is possible that the communication triggers *too many* ready-to-execute computation blocks that should *not* be executed immediately, which otherwise would lead to situations that the next communication step cannot overlap the all ready-to-execute computation.

To solve this problem, we leverage c_Q , c_{KV} , and c_O from profiling and just schedule computation blocks up to these values. Obviously, each GPU should trigger computation blocks in a row-first order, since once a whole row is completed, the output O is ready to be sent.

- **De-prioritizing computations not in critical path.**

Based on the local chunk index notation in Section 3.4, for each GPU, the row corresponding to its local Q has local chunk index 0. This row is also the part of output O that the

Algorithm 2: Mesh-Attention with Greedy Schedule on Each GPU for Forward Pass

```

Required:  $c_Q$ ,  $c_{KV}$ , and  $c_O$ , standing for the least number of block computation to fully hide the time of transferring one Q, KV, or O chunk, respectively

Output: steps

1 Function ComputeBlocks( $x$ ):
2   Compute  $x$  ready-to-execute blocks in a row-first order (but specially, the first row has the lowest priority). If there are fewer than  $x$  objects, compute them all.
3   steps  $\leftarrow []$ , step_num  $\leftarrow 0$ 
4   while Not all Recv Q/KV operations are performed do
5     Set the number of computable blocks unlocked by Recv Q or Recv KV as  $n_Q$  or  $n_{KV}$ 
6     if  $\frac{n_Q}{c_Q} > \frac{n_{KV}}{c_{KV}}$  then
7       steps[step_num++]  $\leftarrow$  "Perform Recv Q, call ComputeBlocks( $c_Q$ ) for overlapping"
8     else
9       steps[step_num++]  $\leftarrow$  "Perform Recv KV, call ComputeBlocks( $c_{KV}$ ) for overlapping"
10  while Not all Send O operations are performed do
11    while Send O is invalid do
12      steps[step_num++]  $\leftarrow$  "call ComputeBlocks(1)"
13    steps[step_num++]  $\leftarrow$  "Perform Send O, call ComputeBlocks( $c_O$ ) for overlapping"
14  while Not all blocks are computed do
15    steps[step_num++]  $\leftarrow$  "ComputeBlocks(1)"

```

GPU is responsible, i.e., it does not send partial O to other GPUs in the same Q group but just receives the reduced O from the predecessor in the group and performs the final reduction. In another word, the computation in this row is *not in the critical path waited by other GPUs*. For this reason, this row should be de-prioritized when necessary.

Based on the three principles, the complete algorithm is described in Algorithm 2. The `ComputeBlocks` function triggers at most x computation blocks following row-first order. The whole attention operation is roughly divided into three parts: (1) scheduling all Recv Q/KV operations aiming to maximizing “profit”, making all computation blocks ready-to-execute (line 4 to line 9); (2) if still some partial outputs are not computed, trigger computation blocks in the corresponding rows to produce outputs, send it to other GPUs, and schedule proper number of computation blocks to be overlapped (line 10 to line 13); and (3) if the local output has not been produced, finish it (line 14 to line 15). Figure 5 (e) provides a running example of the algorithm in action for the 9-GPU example.

The greedy algorithm produces the schedule for a specific shape of tile given a and b . But for a given $n = a \times b$, there are multiple choices of a and b , we need to profile c_Q , c_{KV} , and c_O for each setting, generate the corresponding best schedule based on the greedy algorithm, estimate the runtime, and choose the shape and schedule with the smallest runtime. The flow is shown in Figure 6.

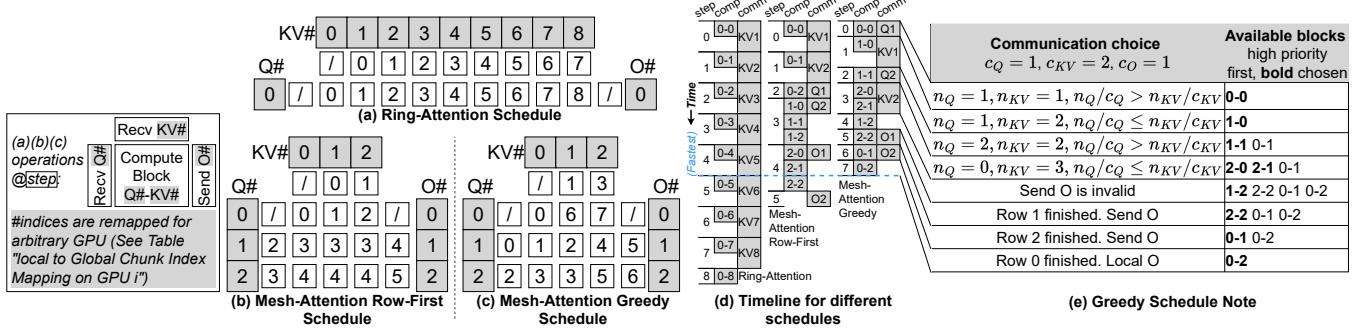


Figure 5: Automatic Scheduling Generation: Problems, Insights, and Running Example

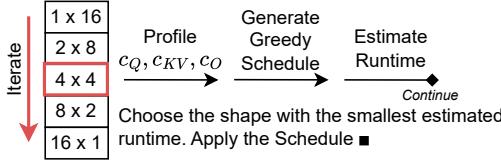


Figure 6: Complete Flow of Scheduling Generation

3.6 Backward Pass of Mesh-Attention

To implement the backward pass, Mesh-Attention needs to be slightly modified. Following FlashAttention [6]’s recomputation strategy, the input chunks transmitted within each Q group consist of the forward output O , its gradient dO , the query tensor Q and the log-sum-exp coefficients lse (with mathematical symbols denoted as $OdOQ$). Meanwhile, the chunks transmitted within each KV group remain the KV tensors (KV). After the computation, the gradients of the non-local KV tensors: dK and dV (collectively denoted as dKV) chunks (dKV), must be sent back along the KV group for reduction on their corresponding GPUs, while the gradients of Q tensor: dQ chunks (dQ), are reduced along the Q group. Therefore, we need four types of P2P communication: **Recv OdOQ/KV** and **Send dQ/dKV**.

The computation on $OdOQ\#i - KV_j$ becomes ready-to-execute after $i - 1$ **Recv Q** operations and $j - 1$ **Recv KV** operations have been performed in prior steps. As for the gradients, the i -th **Send dQ** operation (starting from 1-st) can only be performed after a whole row of attention computations for $\{OdOQ\#i - KV\#j\}_{j=0}^{b-1}$ have finished in prior steps. Similarly, the i -th **Send dKV** operation can only be performed after a whole column of attention computations for $\{OdOQ\#i - KV\#j\}_{i=0}^{a-1}$ have finished in prior steps.

We profile c_{OdOQ} , c_{KV} , c_{dQ} , and c_{dKV} to represent the least number of block computations to fully hide the time to transfer one $OdOQ$, KV , dQ or dKV chunk, respectively, and follow the similar schedule principles as in the forward pass. However, regarding the computation order, a simple row-first order, which is used in the forward pass, can no longer be adopted, as it would defer all dKV chunks transmissions to the end, preventing sufficient overlap with computation; likewise, a column-first order would postpone all dQ chunks transmis-

sions to the end. This challenge stems from the fact that both the Q (row) and KV (column) dimensions involve gradients that must be sent.

To solve this problem, we need a scheme that alternates between finishing rows and columns, which can be achieved through a greedy method. When selecting the next block to compute, there are two candidates: the row-first block and the column-first block. We suppose that after computing n_{dQ} more blocks along the row, or n_{dKV} more blocks along the column, a dQ/dKV chunk can be sent. Since the total computation time is fixed, communication should be initiated as early as possible to maximize overlap. Thus, we compare the profit by comparing $\frac{c_{dQ}}{n_{dQ}}$ and $\frac{c_{dKV}}{n_{dKV}}$: the larger one indicates that a gradient chunk can be sent earlier, weighted by the cost for sending. Moreover, since choosing a row-first or column-first block reduces n_{dQ} or n_{dKV} by one (unless it reaches zero and starts the next row/column), the greedy choice creates a *positive feedback effect*: whenever possible, it will naturally continue until finishing the computation of an entire row or column, satisfying the heuristics of alternately finishing rows and columns.

The details are presented in Algorithm 3. The **ComputeBlocks** function triggers at most x computation blocks, with the **ChooseNextBlock** function deciding the next block to compute following idea of alternating between finishing rows and columns. The whole attention operation is divided into two parts: (1) scheduling all **Recv OdOQ/KV** operations aiming to maximizing profit, making all computation blocks ready-to-execute (line 14 to line 19); (2) if still some partial gradients are not computed, trigger computation blocks to finish either a row or a column to produce a dQ or dKV chunk, send it to the other GPUs, and schedule proper number of computation blocks to be overlapped (line 20 to line 26).

3.7 Causal Mask Mechanism

Causal mask ensures that each token can only attend to itself and tokens that precede it in the sequence, which enforces the left-to-right dependency constraint required for autoregressive generation. Ring-Attention can be adapted to support causal

Algorithm 3: Mesh-Attention with Greedy Schedule on Each Node for Backward Pass

Required: c_{OoQ} , c_{KV} , c_{dQ} , and c_{dKV} , standing for the least number of block computation to fully hide the time of transferring one OdOO, KV, dQ or dKV chunk, respectively

Output : steps

- 1 **Function** ChooseNextBlock():
 - 2 $n_{dQ} \leftarrow$ the number of non-executed blocks in the first unfinished row
 - 3 $n_{dKV} \leftarrow$ the number of non-executed blocks in the first unfinished column
 - 4 **if** $\frac{c_{dQ}}{n_{dQ}} < \frac{c_{dKV}}{n_{dKV}}$ **then**
 - 5 **return** the next ready-to-execute block in column-first order
 - 6 **else**
 - 7 **return** the next ready-to-execute block in row-first order
- 8 **Function** ComputeBlocks(x):
 - 9 **if** there are fewer than x ready-to-execute blocks **then**
 - 10 Compute them all
 - 11 **else**
 - 12 Do “call ChooseNextBlock() and compute the returned block” for x times
- 13 steps $\leftarrow []$, step_num $\leftarrow 0$
- 14 **while** Not all Recv OdOO/KV operations are performed **do**
- 15 Set the number of computable blocks unlocked by Recv OdOO or Recv KV as n_{OoQ} or n_{KV}
- 16 **if** $\frac{n_{OoQ}}{c_{OoQ}} > \frac{n_{KV}}{c_{KV}}$ **then**
- 17 steps[step_num++] \leftarrow “Perform Recv OdOO, call ComputeBlocks(c_{OoQ}) for overlapping”
- 18 **else**
- 19 steps[step_num++] \leftarrow “Perform Recv KV, call ComputeBlocks(c_{KV}) for overlapping”
- 20 **while** Not all Send dQ/dKV operations are performed **do**
- 21 **while** Send dQ/dKV is invalid **do**
- 22 steps[step_num++] \leftarrow “call ComputeBlocks(1)”
- 23 **if** Send dQ is valid **then**
- 24 steps[step_num++] \leftarrow “Perform Send dQ, call ComputeBlocks(c_{dQ}) for overlapping”
- 25 **if** Send dKV is legal **then**
- 26 steps[step_num++] \leftarrow “Perform Send dKV, call ComputeBlocks(c_{dKV}) for overlapping”

attention with Striped Attention [4], which arranges QKV chunks in a striped pattern across GPUs, or Zig-Zag Attention [27], which leverages a zig-zag pattern that pairs slices from the beginning and end of the sequence in opposite directions before distributing them across GPUs.

Mesh-Attention can naturally support causal mask by applying Striped Attention. First, each chunk collects the data mapped from all tokens whose indices share the same remainder modulo n . For example, Q_i / KV_i contains the query / key&value projections of tokens $\{i + nx\}_{x=0}^{N/n-1}$. Then, we adjust the mapping of KV chunk to GPUs to preserve the locality property using the method discussed earlier. Figure 7 shows an example of this procedure with 16 tokens and 4 chunks, after the first step, KV1 and KV2 are swapped so that each GPU can perform computation on its local Q and KV

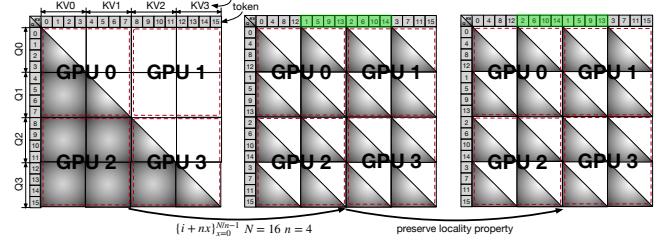


Figure 7: Mesh-Attention with Causal Mask

chunks. We can see that every computation block strictly follows the causal mask and the computation is balanced among the GPUs. Note that Zig-Zag Attention can also be adapted to Mesh-Attention, we can generate the balanced Q/KV chunk partition among GPUs with balanced computation, but due to the complexity of the process, we do not provide the details in this paper.

3.8 Theoretical Communication Complexity Analysis

Table 2: Theoretical Communication Volume (Forward) in Different Sequence-Parallel Methods

Method	Communication Volume Approximate	Best Parallel Limits
Ring-Attn	$(2 - 2/n)Nd \approx 2Nd$	$\downarrow 2Nd$ \uparrow Unlimited
Ulysses	$4(n-1)/n^2Nd \approx 4/nNd$	$\uparrow\uparrow\uparrow 4/nNd$ \downarrow #Heads
StarTrail	$((4C-4)/n + 2/C)Nd$	$\uparrow \approx 4\sqrt{2/n}Nd$ \uparrow Unlimited $(C = \sqrt{n/2})$
Mesh-Attn	$(2a/n + 2/a - 4/n)Nd$	$\uparrow\uparrow \approx 4\sqrt{1/n}Nd$ \uparrow Unlimited $(a = \sqrt{n})$

We present the communication complexity analysis for Ring-Attention, DS-Ulysses, StarTrail, and Mesh-Attention. We assumed that the batch size is 1, the sequence length is N , the parallel degree (i.e., the number of GPUs) is n , and the hidden size is d . Due to space limit, we only discuss the forward pass.

- **Ring-Attention** [14]. Each GPU sends $n - 1$ KV chunks, each of size $2Nd/n$, to the successor in the ring, resulting in a communication volume of $(2 - 2/n)Nd$;

- **DS-Ulysses** [13]. The forward pass of DS-Ulysses contains 4 all-to-all operations of Q, K, V, and O chunks, each counted as $(n - 1)/n^2Nd$. The per-GPU communication volume is thus $4(n - 1)/n^2Nd$;

- **StarTrail** [16]. C is a configurable hyper-parameter of StarTrail, denoting the *attention parallel size* which ranges from 1 to \sqrt{n} . The communication is divided into four stages: an all-gather stage with $3(C - 1)dN/n$ per-GPU communication, a K&V initialization stage with $2CdN/n$ per-GPU communication, a P2P communication stage with $2(n/C - C)Nd/n$ communication volume, and a reduce-scatter stage

with $(C - 1)dN/n$ per-GPU communication size. Taken together, the forward pass of StarTrail requires each GPU to handle the total communication volume up to $((4C - 1)/n + 2/C)Nd$ elements.

- **Mesh-Attention.** The per-GPU communication volume consists of receiving $a - 1$ Q chunks, receiving $n/a - 1$ KV chunks, and sending $a - 1$ O chunks. Thus, the communication volume of each GPU is calculated by the equation below.

$$\begin{aligned} & (a - 1)Nd/n + 2(n/a - 1)Nd/n + (a - 1)Nd/n \\ & = (2a/n + 2/a - 4/n)Nd \end{aligned}$$

By the arithmetic-geometric mean inequality (AM-GM), per-GPU communication is minimized when $a \rightarrow \sqrt{n}$, which leads to an optimal communication volume of $(4\sqrt{1/n} - 4/n)Nd$. In practice, the performance of Mesh-Attention is determined by multiple factors like the scheduling and hardware characteristics. Hence, the chosen factorization might not be the one that has the lowest theoretical communication complexity (i.e., $a = \sqrt{n}$).

In Table 2, we compare the theoretical communication volumes of these four sequence parallelism methods.

StarTrail has a worse communication complexity comparing to Mesh-Attention because of its redundant communication: some GPUs receive KV chunks that are never used for their own computation but are instead relayed to other GPUs. In contrast, the communication structure of Mesh-Attention ensures that every Q or KV chunk received by a GPU is fully utilized for computation, thereby eliminating redundant communication. Moreover, as for computation and communication overlapping, StarTrail’s multi-phase workflow presents certain drawbacks. Its all-gather operation can only overlap with the QKV projection, while the K&V initialization and reduce-scatter operations lack corresponding computations to overlap with. These operations incur considerable communication volume, which can negatively impact overall performance. Beyond communication complexity, DS-Ulysses is constrained by the number of heads, leading to limited parallelism.

4 Experiments

4.1 Experimental Settings

We evaluate Mesh-Attention on a cluster equipped with 256 GPUs unless otherwise specified. For the attention configuration, we choose 32 for the number of heads and 128 for the head dimension, result in a total hidden size of 4096.

4.2 Overall Performance

We compare the runtime and MFU of Mesh-Attention with Ring-Attention in Table 3 and Table 4 with various sequence lengths and number of GPUs. Mesh-Attention consistently

Table 3: Attention Forward + Backward throughput (Unit 10^{-2} iter/s)

Causal Seqlen	Ring-Attention				Mesh-Attention(speedup)				
	32	GPU	64	128	256	32	64	128	256
Y 256k	13.6	10.1	9.0	8.0	35.4(2.6x)	33.3(3.3x)	26.1(2.9x)	22.1(2.7x)	
Y 512k	6.5	5.1	4.9	4.4	15.8(2.5x)	16.9(3.3x)	15.8(3.2x)	11.8(2.7x)	
Y 1M	2.3	2.7	2.6	2.3	5.5(2.4x)	7.3(2.8x)	8.4(3.3x)	7.8(3.4x)	
N 256k	10.9	10.3	8.9	7.8	30.3(2.8x)	31.7(3.1x)	27.9(3.1x)	21.2(2.7x)	
N 512k	4.9	4.9	4.9	4.5	11.1(2.3x)	14.7(3.0x)	13.8(2.8x)	13.9(3.1x)	
N 1M	2.1	2.4	2.5	2.3	3.6(1.7x)	5.7(2.4x)	6.8(2.8x)	6.8(2.9x)	

Table 4: Attention Model FLOPs Utilization (MFU) (%)

Causal Seqlen	Ring-Attention				Mesh-Attention				
	32	GPU	64	128	256	32	64	128	256
Y 256k	6.0	2.2	1.0	0.4	15.6(+9.6)	7.4(+5.2)	2.9(+1.9)	1.2(+0.8)	
Y 512k	11.4	4.5	2.2	1.0	28.0(+16.6)	15.0(+10.5)	7.0(+4.8)	2.6(+1.6)	
Y 1M	16.3	9.4	4.6	2.1	38.9(+22.6)	25.9(+16.5)	14.9(+10.3)	6.9(+4.8)	
N 256k	9.6	4.5	2.0	0.9	26.7(+17.1)	14.0(+9.5)	6.2(+4.2)	2.3(+1.4)	
N 512k	17.3	8.7	4.4	2.0	39.2(+21.9)	25.9(+17.2)	12.1(+7.7)	6.2(+4.2)	
N 1M	29.9	16.6	8.7	4.1	51.0(+21.1)	40.2(+23.6)	24.2(+15.5)	12.1(+8.0)	

outperforms Ring-Attention in various settings and achieves a speedup of up to $3.4\times$ (on average $2.9\times$). Mesh-Attention also consistently achieves a better MFU (model FLOPs utilization). On average, Mesh-Attention’s MFU is $2.5\times$ higher than Ring-Attention (up to $3.4\times$). For both Mesh-Attention and Ring-Attention, MFU becomes higher as the sequence length increases. It is because the computation of attention increases quadratically w.r.t. sequence length while communication only increases linearly. Also, MFUs with causal mask are usually lower since causal mask reduces the computation by half, which makes communication a more severe bottleneck.

4.3 Scalability

Strong Scalability. Figure 8(a) shows the strong scalability results of Mesh-Attention and Ring-Attention. We fix the sequence length to 1 million tokens and varies the number of GPUs used to parallelize the attention operator. Mesh-Attention exhibits a better scalability comparing to Ring-Attention: Ring-Attention can only scale to 64 GPUs and its performance degrades significantly when more GPUs are used. In comparison, Mesh-Attention can scale to 128 GPUs thanks to its lower communication complexity. The best execution time of Ring-Attention is achieved on 64 GPUs (37.5 seconds), which is $3.15\times$ slower than that of Mesh-Attention achieved on 128 GPUs (11.9 seconds).

Weak Scalability. We also conduct weak scalability analysis as shown in Figure 8(b). We vary the number of GPUs and scale the sequence length accordingly so that the amount of computation per GPU remained the same. Since the computation of attention increases quadratically w.r.t. the sequence length, we only increase the sequence length by $\sqrt{2}\times$ when the number of GPUs is doubled. We observed that the runtime of Ring-Attention increases much faster when scaling to a

larger scale because of their higher communication overhead. For Ring-Attention, the execution time of 256 GPUs is $3.74 \times$ slower than that of 32 GPUs. In contrast, Mesh-Attention is only $2.83 \times$ slower.

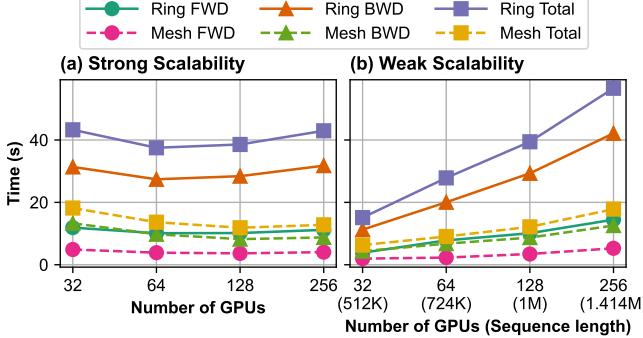


Figure 8: Attention Strong Scaling of Sequence Length 1M and Weak Scaling of Sequence Length from 512K to 1.414M (with Causal Mask)

4.4 Execution Time Breakdown Analysis

We present the execution time breakdown analysis in Figure 9a, which compares the computation time (FWD/BWD Comp) and communication time that is not hidden by computation (FWD/BWD Wait) of Mesh-Attention and Ring-Attention. It confirms that the performance advantage of Mesh-Attention is mostly because of the reduction in communication overhead: the computation time of Mesh-Attention and Ring-Attention is almost the same, while the communication waiting time of Mesh-Attention is up to 74.9% (on average 74.0%) less than that of Ring-Attention. It is also worth noting that, although significantly reduced, the communication overhead of Mesh-Attention is still non-negligible. With 256 GPUs, the communication overhead still takes up 86.6% of the total execution time. We leave the investigation of more advanced techniques to reduce and hide the communication cost as our future work.

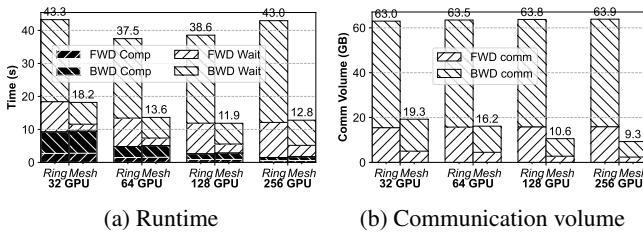


Figure 9: Runtime and Communication Volume Breakdown of Sequence Length 1M for Each GPU (with Causal Mask)

4.5 Communication Analysis

We also present the communication volume (per-GPU) in Figure 9b. Mesh-Attention is able to reduce the communication

Table 5: Attention Forward / backward Peak Memory (GB)

Causal Seqen	Ring-Attention				Mesh-Attention			
	32 GPU	64	128	256	32	64	128	256
Y 256k	1.0/1.5	0.5/0.8	0.3/0.4	0.2/0.2	4.1/7.9	2.2/4.4	2.5/2.5	1.7/2.9
Y 512k	2.1/3.1	1.0/1.5	0.5/0.8	0.3/0.4	8.2/15.9	8.7/8.7	2.4/5.0	3.4/6.8
Y 1M	4.1/6.2	2.1/3.1	1.0/1.5	0.5/0.8	10.3/22.6	7.2/15.9	4.9/10.0	3.2/8.1
N 256k	1.0/1.5	0.5/0.8	0.3/0.4	0.1/0.2	4.1/7.9	2.2/4.4	1.2/2.5	1.0/3.4
N 512k	2.1/3.1	1.0/1.5	0.5/0.8	0.3/0.4	5.1/11.3	3.5/8.7	2.4/5.0	2.4/4.0
N 1M	4.1/6.2	2.1/3.1	1.0/1.5	0.5/0.8	2.0/32.1	7.2/17.7	4.9/10.0	3.0/7.3

volume by up to 85.5% (on average 78.2%). It is worth noting that when scaling the number of GPUs, the communication volume of Ring-Attention remains almost the same while that of Mesh-Attention decreases significantly. For example, per-GPU communication volume of Mesh-Attention with 256 GPUs is 51.8% less than that with 32 GPUs. The observation is consistent with our theoretical analysis in Section 3.8, which demonstrates the better scale-out communication property of Mesh-Attention.

4.6 Peak Memory Analysis

We analyze the peak memory consumption of Mesh-Attention in Table 5. In general, Mesh-Attention consumes more peak memory than Ring-Attention since it needs to cache multiple KV/Q chunks throughout the entire attention computation for data reuse. In contrast, Ring-Attention keeps at most 2 KV chunk and 1 Q chunk in the memory. However, it is worth noting that the high peak memory consumption of Mesh-Attention is transient: most of the GPU memory will be quickly released once the attention forward/backward computation of the current layer completes. Mesh-Attention will not increase the amount of stashed forward activations that are needed by the backward pass. We leave exploring memory-efficient schedules (e.g., schedules that release KV/Q chunks in a more eager manner) as our future work.

4.7 Evaluation with GQA

Besides standard multi-head attention (MHA), we also evaluate the performance of Mesh-Attention for grouped-query attention (GQA) [2], an attention variant that slightly trades model expressiveness for better efficiency. GQA reduces the number of KV heads by a factor of g , and allows consecutive g Q heads to share the same KV head for attention. As g increases, the size of the KV tensor decreases and hence the communication bottleneck of Ring-Attention is alleviated. We compare the performance of Mesh-Attention and Ring-Attention in Figure 10 with various g . Mesh-Attention consistently outperforms Ring-Attention in all settings. When g is large (e.g., 4/8), Mesh-Attention still greatly reduces the communication overhead. However, since the communication problem is less severe for Ring-Attention, the overall performance gains of Mesh-Attention is less significant.

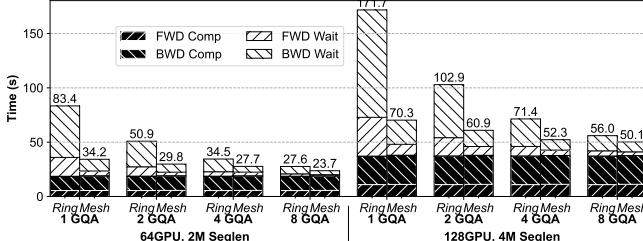


Figure 10: Runtime Breakdown of Different GQA Degrees

5 Conclusion

This paper proposes Mesh-Attention, a new distributed attention algorithm that assigns a two-dimensional tile—rather than one-dimensional row or column—of computation blocks to each GPU to achieve higher efficiency through lower communication-computation (CommCom) ratio. The general approach covers Ring-Attention as a special case, and allows the tuning of CommCom ratio with different tile shapes. Importantly, we propose a greedy algorithm that can efficiently search the scheduling space within the tile with restrictions that ensure efficient communication among GPUs. The theoretical analysis shows that Mesh-Attention leads to a much lower communication complexity and exhibits good scalability comparing to other current algorithms. The experimental results convincingly confirm the advantage of Mesh-Attention.

References

- [1] The claudie 3 model family: Opus, sonnet, haiku.
- [2] 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, 2023.
- [3] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2024.
- [4] William Brandon, Aniruddha Krusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers, 2023.
- [5] Zesen Cheng, Sicong Leng, Hang Zhang, Yifei Xin, Xin Li, Guanzheng Chen, Yongxin Zhu, Wenqi Zhang, Ziyang Luo, Deli Zhao, and Lidong Bing. Videollama 2: Advancing spatial-temporal modeling and audio understanding in video-lmms, 2024.
- [6] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.
- [7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.
- [8] Jiarui Fang and Shangchun Zhao. Usp: A unified sequence parallelism approach for long context generative ai, 2024.
- [9] Google DeepMind. Gemini 2.5: Our most intelligent ai model yet, 2025. Accessed: 2025-10-13.
- [10] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2018.
- [11] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.
- [12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, Hyoukjoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.
- [13] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.
- [14] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023.
- [15] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents, 2025.
- [16] Ziming Liu, Shaoyu Wang, Shenggan Cheng, Zhongkai Zhao, Kai Wang, Xuanlei Zhao, James Demmel, and Yang You. Startrail: Concentric ring sequence parallelism for efficient near-infinite-context transformer model training, 2025.
- [17] Meta AI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation, 2025. Accessed: 2025-10-13.
- [18] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax, 2018.

- [19] OpenAI. Gpt-4 technical report. Technical report, OpenAI, 2023.
- [20] Mohammad Shoeybi, Mostafa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.
- [21] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs.experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery.
- [22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.
- [23] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. Context parallelism for scalable million-token inference, 2025.
- [24] Hang Zhang, Xin Li, and Lidong Bing. Video-llama: An instruction-tuned audio-visual language model for video understanding, 2023.
- [25] Yang Zhang, Hanlei Jin, Dan Meng, Jun Wang, and Jinghua Tan. A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods, 2025.
- [26] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2025.
- [27] Zilin Zhu. Feature request: Balancing computation with zigzag blocking. <https://github.com/zhuzilin/ring-flash-attention/issues/2>, 2024. Accessed: 2025-10-13.