Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

# Memory-efficient tensor parallelism for long-sequence Transformer training*

Peng LIANG, Linbo QIAO, Yanqi SHI, Hao ZHENG, Yu TANG, Dongsheng LI†‡

*National Key Laboratory of Parallel and Distributed Computing, College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China*

†E-mail: dsli@nudt.edu.cn

Received July 17, 2024; Revision accepted Feb. 23, 2025; Crosschecked Mar. 7, 2025; Published online Apr. 2, 2025

**Abstract:** Transformer-based models like large language models (LLMs) have attracted significant attention in recent years due to their superior performance. A long sequence of input tokens is essential for industrial LLMs to provide better user services. However, memory consumption increases quadratically with the increase of sequence length, posing challenges for scaling up long-sequence training. Current parallelism methods produce duplicated tensors during execution, leaving space for improving memory efficiency. Additionally, tensor parallelism (TP) cannot achieve effective overlap between computation and communication. To solve these weaknesses, we propose a general parallelism method called memory-efficient tensor parallelism (METP), designed for the computation of two consecutive matrix multiplications and a possible function between them ($\boldsymbol{O} = f(\boldsymbol{AB})\boldsymbol{C}$), which is the kernel computation component in Transformer training. METP distributes subtasks of computing $\boldsymbol{O}$ to multiple devices and uses send/recv instead of collective communication to exchange submatrices for finishing the computation, avoiding producing duplicated tensors. We also apply the double buffering technique to achieve better overlap between computation and communication. We present the theoretical condition of full overlap to help instruct the long-sequence training of Transformers. Suppose the parallel degree is $p$; through theoretical analysis, we prove that METP provides $O(1/p^3)$ memory overhead when not using FlashAttention to compute attention and could save at least 41.7% memory compared to TP when using FlashAttention to compute multi-head self-attention. Our experimental results demonstrate that METP can increase the sequence length by 2.38–2.99 times compared to other methods when using eight A100 graphics processing units (GPUs).

**Key words:** Distributed learning; Large language model (LLM); Long sequence; Machine learning system; Memory efficiency; Tensor parallelism

https://doi.org/10.1631/FITEE.2400602 **CLC number:** TP183

## 1 Introduction

Training language models with long sequences is crucial for handling long-context tasks such as document summarization, dialogue systems, and technical writing (Beltagy et al., 2020; Kaddour et al., 2023; Tarassow, 2023). This is because the context length determines the maximum word count a language model can handle, and longer sequences improve contextual understanding and semantic coherence, allowing models to generate more coherent and contextually appropriate text (Achiam et al., 2023; Huang et al., 2023). However, it requires $O(s^2)$ memory for a sequence with $s$ tokens to complete the attention computation in Transformer models. This memory limitation makes it challenging for language models to calculate extremely long sequences during training. Additionally, training with long sequences costs $O(s^2)$ time for attention computation, bringing

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (Nos. 62025208 and 62421002)

ORCID: Peng LIANG, https://orcid.org/0000-0002-5590-5179; Dongsheng LI, https://orcid.org/0000-0001-9743-2034

computation challenges to a single device.

To address these challenges, researchers proposed various techniques such as tensor parallelism (TP) (Narayanan et al., 2021a), fully sharded data parallelism (FSDP) (Zhao et al., 2023), sequence parallelism (SP) (Li SG et al., 2023; Jacobs et al., 2024), and pipeline parallelism (PP) (Huang et al., 2019; Narayanan et al., 2021b; Liu ZM et al., 2023) to distribute computation and memory costs across multiple devices. These techniques can reduce the memory cost of training large language models (LLMs) to $O(1/p)$, where $p$ denotes the parallel degree. Since these methods are orthogonal to each other, using multiple parallelism methods simultaneously, such as 3D parallelism (Narayanan et al., 2021a; Lai et al., 2023), can reduce the memory footprint to $O(1/(p_\mathrm{t}p_\mathrm{d}p_\mathrm{p}))$, where $p_\mathrm{t}$, $p_\mathrm{d}$, and $p_\mathrm{p}$ represent the parallel degrees of TP, data parallelism (DP), and PP, respectively. However, 3D parallelism could only reduce memory footprint to $\Omega(1/\#\mathrm{device})$. This work aims to find a 1D parallelism method that can achieve memory usage lower than $\Omega(1/\#\mathrm{device})$.

FlashAttention (Dao et al., 2022) achieves high memory efficiency by employing tiling techniques to optimize the attention computation process, which comprises two matrix multiplications interleaved with a Softmax operation. This method divides the query, key, and value matrices into submatrices and calculates attention scores for each query submatrix by iterating over all the key and value submatrices, avoiding the significant memory overhead of storing the query@key. As a result, it reduces the memory footprint of computing attention from $O(s^2)$ to $O(s)$, where $s$ represents the sequence length.

Motivated by FlashAttention, we propose an approach called memory-efficient tensor parallelism (METP), which focuses on reducing the memory footprint of computation with a form of $\boldsymbol{O} = f(\boldsymbol{AB})\boldsymbol{C}$ by distributing and scheduling the subtasks across devices. METP can reduce the memory usage of computing attention for long sequences to $O(1/p^3)$. Additionally, it is orthogonal to DP and PP.

The main contributions of this paper are three-fold:

1. We propose a method named METP, which decomposes the computation tasks by tiling to handle the large memory overhead problem of training LLMs with long sequences.

2. We present a detailed theoretical analysis of the memory and communication overhead of METP and a detailed comparison with TP, FSDP, and SP. We investigate the conditions to achieve the best performance for METP and prove that METP can reduce the memory consumption of attention scores to $1/p^3$ when not using FlashAttention and other intermediate results by at least 41.7%, where $p$ is the parallel degree of METP.

3. Experimental results demonstrate that METP can achieve memory savings ranging from 8% to 70% for sequences longer than 32k while preserving computational efficiency.

## 2 Background and related works

### 2.1 Components of memory consumption in training LLMs

Memory usage in training neural networks with backward propagation algorithms consists of two main components: model states and intermediate results (Rajbhandari et al., 2020). Model states include the model parameters, gradients of the model parameters, and optimizer states. When training with the Adam optimizer (Kingma and Ba, 2015), a widely used optimizer in LLM training, the memory required for model states is approximately four times the size of the model. Intermediate results are generated during the forward and backward propagation. Some intermediate results participate in the computation of backward propagation and thus must be stored until used. During execution, devices need to have enough memory to store the model states and the generated intermediate results. Therefore, peak memory usage often occurs when forward propagation is about to end or when backward propagation starts.

As the latest models scale towards billions of parameters, parallel and distributed training is required to divide the model states and intermediate results across computing devices so that they fit into each graphics processing unit (GPU)'s memory and enable training within a realistic time.

### 2.2 FlashAttention

To train Transformers faster, Dao et al. (2022) proposed FlashAttention, an input/output (IO)-aware exact attention algorithm that uses tiling to

reduce the number of reads/writes between GPU's high bandwidth memory (HBM) and GPU's on-chip static random-access memory (SRAM). This approach exploits the asymmetric GPU memory hierarchy to bring significant memory savings and speedups compared to optimized baselines. Additionally, it reduces the memory consumption of attention computation to $O(s)$, demonstrating great potential for training long-context tasks. Furthermore, Dao (2024) proposed FlashAttention-2 to achieve better parallelism and work partitioning. This is done by further optimizing the work partitioning between different thread blocks and warps on the GPU, resulting in a 2× speedup compared to FlashAttention on the Ampere architecture. FlashAttention-3 (Shah et al., 2024), tailored for the Hopper architecture, enhances the utilization rate from 35% to 75% on the H100 GPU. Inspired by FlashAttention-2, we distribute submatrices of $\boldsymbol{K}$ and $\boldsymbol{V}$ across different devices to achieve memory efficiency.

### 2.3 Intra-operator parallelism

As shown in Fig. 1, intra-operator parallelism methods, such as FSDP, SP, and TP, distribute tensors within an operator across several devices. They usually result in a larger communication volume than inter-operator parallelism and thus are often applied to devices connected with high-speed interconnects (Liang et al., 2023).

#### 2.3.1 FSDP

FSDP, an implementation of ZeRO (Rajbhandari et al., 2020), is a method aimed at optimizing the memory footprint of model states. By partitioning the model states, it achieves an $O(1/p)$ memory cost while retaining low communication volume and high computational granularity. Due to FSDP's significant memory savings, these preserved resources enable training LLMs with extended input sequences. However, FSDP can only shard the data along the batch-size axis, which limits its ability to handle very long sequences.

#### 2.3.2 SP

Initially, SP divides a long input sequence into several subsequences and distributes them across different GPUs.

Li SG et al. (2023) applied a ring self-attention (RSA) algorithm to use the tokens from different ranks. Although this approach partitions the sequences, it still needs to store the complete $O(s^2)$ results of query@key, resulting in poor scalability.

To address the issue of ZeRO/FSDP's inability to process long sequences, Jacobs et al. (2024) proposed Ulysses, which uses AllToAll to transpose the tensors before and after attention computation, making FSDP partition inputs along the sequence axis.

Motivated by the RSA algorithm, we design an enhanced ring algorithm and apply it to METP to handle subtasks in computing attention.

#### 2.3.3 TP

Narayanan et al. (2021a) proposed TP, which partitions model states and activations to improve training efficiency. Subsequently, they integrated SP (Korthikanti et al., 2023) into their framework. Different from the vanilla SP (Li SG et al., 2023), their SP employs the communication methods of ReduceScatter and AllGather, both before sequence partitioning and after computation completion. Additionally, parallel implementations are applied to Dropout (Srivastava et al., 2014) layers and LayerNorm (Xu et al., 2019) layers. To simplify the explanation, we assume that the TP from Megatron-LM works in conjunction with this specific form of SP (Korthikanti et al., 2023).

In Section 3, we provide the overhead analysis of the abovementioned intra-operator parallelism methods to compare with METP.

## 3 METP

We develop a general method called METP for the computation of $\boldsymbol{O} = f(\boldsymbol{AB})\boldsymbol{C}$, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are matrices and $f$ is a function consisting of element-wise operators such as Gaussian error linear unit (GeLU) and Dropout. In METP, we simultaneously partition three matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$. Specifically, matrices $\boldsymbol{A}$ and $\boldsymbol{C}$ are divided in a row-wise manner, and matrix $\boldsymbol{B}$ is split in a column-wise manner. METP reduces the memory overhead of intermediate result values while obtaining the correct results by carefully scheduling computation and communication subtasks.

Suppose the parallel degree is $p$, and the shapes

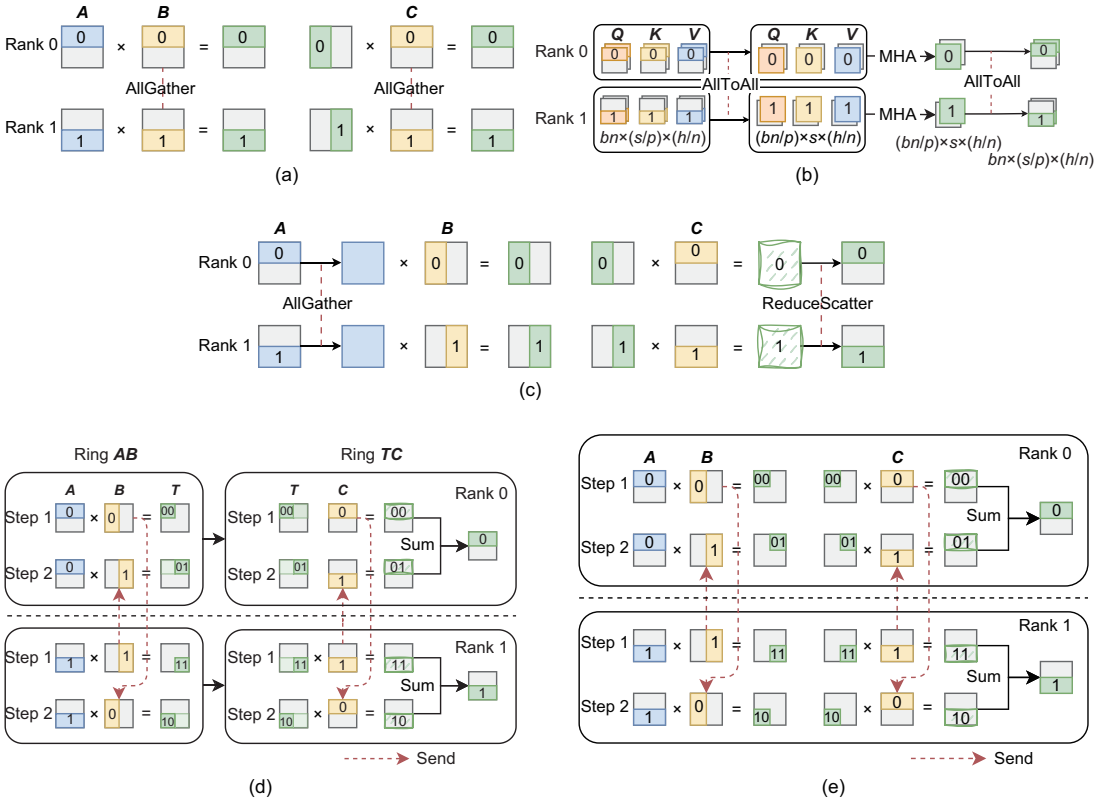


**Fig. 1 Intra-operator parallelism methods: (a) fully sharded data parallelism; (b) Ulysses sequence parallelism; (c) tensor parallelism; (d) ring self-attention sequence parallelism; (e) memory-efficient tensor parallelism (MHA: multi-head self-attention. $b$: batch size; $s$: sequence length; $h$: hidden size; $p$: parallel degree; $n$: number of heads)**

of matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are $m \times k_1$, $k_1 \times k_2$, and $k_2 \times n$, respectively, where $m$, $k_1$, $k_2$, and $n$ are appropriate integers. Subsequently, we split matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ into $p$ submatrices and distribute them to each rank. On rank $i$, submatrix $\boldsymbol{A}_{(i,:)}$ has a shaped of $(m/p) \times k_1$, submatrix $\boldsymbol{B}_{(:,i)}$ assumes a shape of $k_1 \times (k_2/p)$, and submatrix $\boldsymbol{C}_{(i,:)}$ takes the form of $(k_2/p) \times n$. Each submatrix $\boldsymbol{O}_{(i,:)}$ of $\boldsymbol{O}$ is calculated by

$$\boldsymbol{O}_{(i,:)} = \sum_{r=0}^{p-1} f(\boldsymbol{A}_{(i,:)}\boldsymbol{B}_{(:,r)})\boldsymbol{C}_{(r,:)}. \tag{1}$$

Each rank $i$ is responsible for computing the results of $\boldsymbol{O}_{(i,:)}$. Given that the submatrices of $\boldsymbol{B}$ and $\boldsymbol{C}$ are distributed across various ranks, communication is required to acquire them for the computation of $\boldsymbol{A}_{(i,:)}$, $\boldsymbol{B}_{(:,r)}$, and $\boldsymbol{C}_{(r,:)}$. We implement the communication using peer-to-peer (P2P) primitives, which sends $\boldsymbol{B}_{(:,r)}$ and $\boldsymbol{C}_{(r,:)}$ to the next rank and receives $\boldsymbol{B}_{(:,r+1)}$ and $\boldsymbol{C}_{(r+1,:)}$ from the previous rank.

Fig. 1e illustrates the forward execution of METP, where the parallel degree $p$ is 2. In step 1, each rank $i$ calculates the partial result of $\boldsymbol{O}_{(i,:)}$ while concurrently sending its held submatrices of $\boldsymbol{B}$ and $\boldsymbol{C}$. In step 2, the computation is completed using the received submatrices. The general forward and backward passes of METP are detailed in Algorithms 1 and 2, respectively.

Considering the computation of the Transformer, we find that METP can be applied to the calculation of the feed-forward network (FFN):

$$\boldsymbol{Z} = \mathrm{GeLU}(\boldsymbol{X}\boldsymbol{W}_{\mathrm{in}})\boldsymbol{W}_{\mathrm{out}}, \tag{2}$$

where $\boldsymbol{X}$ is the input, $\boldsymbol{Z}$ is the output, and $\boldsymbol{W}_{\mathrm{in}}$ and $\boldsymbol{W}_{\mathrm{out}}$ are the weights corresponding to the in-projection and out-projection layers, respectively.

Taking FFN computation as an example, we now analyze the memory consumption and communication overhead of METP and compare them with those of Megatron-LM, RSA, and FSDP/ZeRO+ Ulysses. We first derive the general form of the overheads using the previously defined notations $m$,

**Algorithm 1** METP forward pass

Prepare $\boldsymbol{A}_i \in \mathbb{R}^{(N/p)\times d_1}$, $\boldsymbol{B}_i^{(0)} \in \mathbb{R}^{d_1\times(d_2/p)}$, and $\boldsymbol{C}_i^{(0)} \in \mathbb{R}^{(d_2/p)\times d_3}$ for rank $i$
Initialize $\boldsymbol{O}_i$
Annotate $r_i^j = (i+j)\%p$
**for** $j = 0$ to $p-1$ **do**
  $\boldsymbol{T}_{ij} \leftarrow \boldsymbol{A}_i\boldsymbol{B}_i^{(j)}$
  $\boldsymbol{D}_{ij} \leftarrow f(\boldsymbol{T}_{ij})$
  $\boldsymbol{O}_i \leftarrow \boldsymbol{O}_i + \boldsymbol{D}_{ij}\boldsymbol{C}_i^{(j)}$
  Send $\boldsymbol{B}_i^{(j)}$ and $\boldsymbol{C}_i^{(j)}$ to rank $r_i^{j-1}$
  Receive $\boldsymbol{B}_{i+1}^{(j)}$ and $\boldsymbol{C}_{i+1}^{(j)}$ from rank $r_i^{j+1}$
  $\boldsymbol{B}_i^{(j+1)} \leftarrow \boldsymbol{B}_{i+1}^{(j)}$, $\boldsymbol{C}_i^{(j+1)} \leftarrow \boldsymbol{C}_{i+1}^{(j)}$
**end for**
Return $\boldsymbol{O}_i$

**Algorithm 2** METP backward pass

Prepare $\boldsymbol{A}_i \in \mathbb{R}^{(N/p)\times d_1}$, $\boldsymbol{B}_i^{(0)} \in \mathbb{R}^{d_1\times(d_2/p)}$, $\boldsymbol{C}_i^{(0)} \in \mathbb{R}^{(d_2/p)\times d_3}$, and $\Delta\boldsymbol{O}_i \in \mathbb{R}^{(N/p)\times d_3}$ for rank $i$
Initialize $\Delta\boldsymbol{A}_i$, $\Delta\boldsymbol{B}_i^{(0)}$, and $\Delta\boldsymbol{C}_i^{(0)}$
Annotate $r_i^j = (i+j)\%p$
**for** $j = 0$ to $p-1$ **do**
  $\boldsymbol{T}_{ij} \leftarrow \boldsymbol{A}_i\boldsymbol{B}_i^{(j)}$
  $\boldsymbol{D}_{ij} \leftarrow f(\boldsymbol{T}_{ij})$
  $\Delta\boldsymbol{D}_{ij} \leftarrow \Delta\boldsymbol{O}_i\left(\boldsymbol{C}_i^{(j)}\right)^{\mathrm{T}}$
  $\Delta\boldsymbol{C}_i^{(j)} \leftarrow \Delta\boldsymbol{C}_i^{(j)} + \boldsymbol{D}_{ij}^{\mathrm{T}}\Delta\boldsymbol{O}_i$
  $\Delta\boldsymbol{T}_{ij} \leftarrow f'(\boldsymbol{T}_{ij})\Delta\boldsymbol{D}_{ij}$
  $\Delta\boldsymbol{A}_i \leftarrow \Delta\boldsymbol{A}_i + \Delta\boldsymbol{T}_{ij}\left(\boldsymbol{B}_i^{(j)}\right)^{\mathrm{T}}$
  $\Delta\boldsymbol{B}_i^{(j)} \leftarrow \Delta\boldsymbol{B}_i^{(j)} + \boldsymbol{A}_i^{\mathrm{T}}\Delta\boldsymbol{T}_{ij}$
  Send $\boldsymbol{B}_i^{(j)}$, $\boldsymbol{C}_i^{(j)}$, $\Delta\boldsymbol{B}_i^{(j)}$, and $\Delta\boldsymbol{C}_i^{(j)}$ to rank $r_i^{j-1}$
  Receive $\boldsymbol{B}_{i+1}^{(j)}$, $\boldsymbol{C}_{i+1}^{(j)}$, $\Delta\boldsymbol{B}_{i+1}^{(j)}$, and $\Delta\boldsymbol{C}_{i+1}^{(j)}$ from rank $r_i^{j+1}$
  $\boldsymbol{B}_i^{(j+1)} \leftarrow \boldsymbol{B}_{i+1}^{(j)}$, $\boldsymbol{C}_i^{(j+1)} \leftarrow \boldsymbol{C}_{i+1}^{(j)}$
  $\Delta\boldsymbol{B}_i^{(j+1)} \leftarrow \Delta\boldsymbol{B}_{i+1}^{(j)}$, $\Delta\boldsymbol{C}_i^{(j+1)} \leftarrow \Delta\boldsymbol{C}_{i+1}^{(j)}$
**end for**
Return $\Delta\boldsymbol{A}_i$, $\Delta\boldsymbol{B}_i$, and $\Delta\boldsymbol{C}_i$

$k_1$, $k_2$, $n$, and $p$. Then, we compute the overhead results for each method applied to the Transformer's FFN layer (i.e., Eq. (2)) and record them in Table 1. Specifically, we set $m = bs$, $k_1 = h$, $k_2 = 4h$, and $n = h$, where $b$, $s$, and $h$ represent the batch size, sequence length, and hidden size, respectively.

### 3.1 Memory consumption analysis

To store the submatrices of $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$, METP needs a total amount of $(mk_1+k_1k_2+k_2n)/p$ memory. Calculating $\boldsymbol{A}_{(i,:)}\boldsymbol{B}_{(:,r)}$ consumes $mk_2/p^2$ intermediate memory. To obtain the final result, we need additional $mn/p$ memory. Finally, the total memory consumption of METP is $(mk_1 + k_1k_2 + k_2n + mn)/p + mk_2/p^2$.

Additionally, if $\boldsymbol{B}$ and $\boldsymbol{C}$ are parameters that require training, additional memory ($3\times$) is needed to store gradients and optimizer states (i.e., momentum and variance) for them. In this case, the memory cost is $(mk_1 + 4k_1k_2 + 4k_2n + mn)/p + mk_2/p^2$.

To prove the memory efficiency of METP, we investigate the memory consumption of other methods. Referring to Li SG et al. (2023), Table 1 lists the memory consumption of Megatron-LM and RSA. For FSDP/ZeRO+Ulysses, each rank stores $1/p$ of the model states and handles inputs with shape $(b, s/p, h)$. Therefore, the static memory consumption is $(32h^2 + 5bsh)/p$. Given that ZeRO needs to collect parameters using the AllGather operator before execution, the peak memory consumption is $4h^2 + (28h^2 + 5bsh)/p$.

According to Table 1, when compared to Megatron-LM, we have

$$\frac{32h^2}{p} + \frac{bsh}{p} + \frac{4bsh}{p^2} < \frac{32h^2}{p} + bsh + \frac{4bsh}{p}. \quad (3)$$

This proves that our method is always more memory-efficient than Megatron-LM. When compared with RSA and FSDP/ZeRO+Ulysses, we have the same conclusion.

### 3.2 Communication overhead analysis

In this subsection, we analyze the communication overheads of METP and other methods for better comparison.

#### 3.2.1 METP

In each iteration of calculating $\boldsymbol{O}_{(i,:)}$, the communication volume generated by P2P communication between the submatrices of $\boldsymbol{B}$ and $\boldsymbol{C}$ equals their size (i.e., $(k_1k_2 + k_2n)/p$). The forward propagation process iterates $p$ times, thus producing a total communication volume of $k_1k_2 + k_2n$. Since the forward pass evicts intermediate results to save memory, we recompute them in the backward pass. Therefore, we must again communicate the submatrices of $\boldsymbol{B}$ and $\boldsymbol{C}$. Additionally, the backward process communicates for the gradients of $\boldsymbol{B}$ and $\boldsymbol{C}$ to accumulate the gradient information for each rank. Thus, the communication volume generated by the backward propagation amounts to $2(k_1k_2 + k_2n)$. Cumulatively, the overall communication volume is $3(k_1k_2 + k_2n)$. Since each rank $i$ already holds $\boldsymbol{B}_{(:,i)}$

**Table 1 Peak memory overhead and communication volume of FFN during training**

| Method | Operation | Shape of matrix 1 | Shape of matrix 2 | Shape of output |
|---|---|---|---|---|
| Megatron-LM (TP+SP) | AG($\boldsymbol{X}$)+$\boldsymbol{X}\boldsymbol{W}_{\text{in}}$ | $b\times(s/p)\times h$ | $h\times(4h/p)$ | $b\times s\times(4h/p)$ |
| | $\boldsymbol{Y}\boldsymbol{W}_{\text{out}}$+RS($\boldsymbol{Z}$) | $b\times s\times(4h/p)$ | $(4h/p)\times h$ | $b\times(s/p)\times h$ |
| RSA | $\boldsymbol{X}\boldsymbol{W}_{\text{in}}$ | $b\times(s/p)\times h$ | $h\times 4h$ | $b\times(s/p)\times 4h$ |
| | $\boldsymbol{Y}\boldsymbol{W}_{\text{out}}$ | $b\times(s/p)\times 4h$ | $4h\times h$ | $b\times(s/p)\times h$ |
| FSDP/ZeRO+Ulysses | AG($\boldsymbol{W}_{\text{in}}$)+$\boldsymbol{X}\boldsymbol{W}_{\text{in}}$ | $b\times(s/p)\times h$ | $(h/p)\times 4h$ | $b\times(s/p)\times 4h$ |
| | AG($\boldsymbol{W}_{\text{out}}$)+$\boldsymbol{Y}\boldsymbol{W}_{\text{out}}$ | $b\times(s/p)\times 4h$ | $(4h/p)\times h$ | $b\times(s/p)\times h$ |
| METP (ours) | P2P($\boldsymbol{W}_{\text{in}}$)+$\boldsymbol{X}\boldsymbol{W}_{\text{in}}$ | $b\times(s/p)\times h$ | $h\times(4h/p)$ | $b\times(s/p)\times(4h/p)$ |
| | P2P($\boldsymbol{W}_{\text{out}}$)+$\boldsymbol{Y}\boldsymbol{W}_{\text{out}}$ | $b\times(s/p)\times(4h/p)$ | $(4h/p)\times h$ | $b\times(s/p)\times h$ |

| Method | Memory | Peak memory | Communication volume |
|---|---|---|---|
| Megatron-LM (TP+SP) | $\frac{32h^2}{p}+\frac{5bsh}{p}$ | $\frac{32h^2}{p}+\frac{4bsh}{p}+bsh$ | $5\frac{p-1}{p}bsh$ |
| RSA | $32h^2+\frac{5bsh}{p}$ | $32h^2+\frac{5bsh}{p}$ | 0 |
| FSDP/ZeRO+Ulysses | $\frac{32h^2}{p}+\frac{5bsh}{p}$ | $4h^2+\frac{28h^2+5bsh}{p}$ | $24\frac{p-1}{p}h^2$ |
| METP (ours) | $\frac{32h^2}{p}+\frac{bsh}{p}$ | $\frac{32h^2}{p}+\frac{bsh}{p}+\frac{4bsh}{p^2}$ | $24\frac{p-1}{p}h^2$ |

$b$: batch size; $s$: sequence length; $h$: hidden size; $p$: parallel degree. Memory: the memory overhead after executing forward propagation. Peak memory: the peak memory overhead during forward propagation. AG: AllGather; RS: ReduceScatter; P2P: peer-to-peer. Note: SP does not produce communication during FFN computation (Li SG et al., 2023)

and $\boldsymbol{C}_{(i,:)}$, we can reduce each P2P communication to $p-1$ times, resulting in a communication volume of $3\frac{p-1}{p}(k_1k_2+k_2n)$. When applying METP to Eq. (2), the communication volume is $24\frac{p-1}{p}h^2$.

#### 3.2.2 Megatron-LM

Megatron-LM employs TP and its version of SP in training. Fig. 1c illustrates that during the forward process, Megatron-LM performs an AllGather operation on $\boldsymbol{A}_{(i,:)}$ to form $\boldsymbol{A}$. Once the computation is complete, ReduceScatter operation is applied to the partial output result of $\boldsymbol{O}$ to obtain $\boldsymbol{O}_{(i,:)}$. For a tensor of size $N$, the communication volume generated by AllGather and ReduceScatter operations is $(p-1)N$ and $\frac{p-1}{p}N$, respectively. Given that the shapes of $\boldsymbol{A}_{(i,:)}$ and $\boldsymbol{O}$ are $(m/p)\times k_1$ and $m\times n$, respectively, the communication volume produced during the forward pass amounts to $\frac{p-1}{p}(mk_1+mn)$. During the backward propagation process, the AllGather operation is executed on $\Delta\boldsymbol{O}_{(i,:)}$ and $\boldsymbol{A}_{(i,:)}$, while ReduceScatter operation is executed on $\Delta\boldsymbol{A}$, leading to a communication volume of $\frac{p-1}{p}(2mk_1+mn)$. Consequently, the overall communication volume for the TP used in Megatron-LM amounts to $\frac{p-1}{p}(3mk_1+2mn)$.

#### 3.2.3 FSDP

As shown in Fig. 1a, FSDP (Zhao et al., 2023) partitions matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ in a row-wise manner. During the forward propagation, FSDP uses two AllGather operations to collect the entire matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ for computation. Once the computation is complete, gathered parameters from other ranks will be evicted. The two AllGather operations generate a communication volume of $\frac{p-1}{p}(k_1k_2+k_2n)$. During the backward propagation, FSDP recollects matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ using an AllGather operator. Additionally, after calculating gradients of $\boldsymbol{B}$ and $\boldsymbol{C}$, FSDP uses a ReduceScatter operator to obtain the complete gradient information for each rank. Thus, the backward propagation process generates a communication volume of $2\frac{p-1}{p}(k_1k_2+k_2n)$. Therefore, the total communication volume for FSDP amounts to $3\frac{p-1}{p}(k_1k_2+k_2n)$.

#### 3.2.4 RSA sequence parallelism

As shown in Fig. 1d, RSA partitions matrices $\boldsymbol{A}$ and $\boldsymbol{C}$ in a row-wise manner and partitions $\boldsymbol{B}$ in a column-wise manner. It also applies P2P communication to send and receive submatrices of $\boldsymbol{B}$ and $\boldsymbol{C}$ from other ranks, which produces a communication volume of $\frac{p-1}{p}(k_1k_2+k_2n)$. After the execution of ring $\boldsymbol{AB}$, during the backward propagation, RSA performs AllReduce on $\Delta\boldsymbol{B}$ and $\Delta\boldsymbol{C}$ and P2P on submatrices of $\boldsymbol{B}$ and $\boldsymbol{C}$, which produces an accumulative communication volume of $3\frac{p-1}{p}(k_1k_2+k_2n)$. Thus, the total communication volume is $4\frac{p-1}{p}(k_1k_2+k_2n)$. RSA only partitions multi-head self-attention (MHA) in this way. For

FFN, it partitions only the input $\boldsymbol{A}$ along the sequence axis and does not generate communication overhead.

#### 3.2.5 Ulysses SP

Like RSA, Ulysses incurs communication overhead only during the computation of attention. It employs an AllToAll operator to reorganize the layout of tensors $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$, transforming them from a sequence-axis partition to a head-num-axis partition. After attention computation, another AllToAll operation is applied to switch back to the sequence-axis partition layout. During the backward propagation, there are also two AllToAll operators. Overall, the communication overhead of Ulysses is $8\frac{p-1}{p^2}bsh$.

We summarize the communication volume of different methods applied to FFN in Table 1 and find that our method achieves the same communication cost as FSDP/ZeRO+Ulysses during the training of Transformer's FFN layers while offering better memory efficiency. It is important to note that RSA does not produce any communication during FFN computation, according to Li SG et al. (2023).

### 3.3 Overlap between computation and communication

Eq. (1) reveals that $\boldsymbol{O}_{(i,:)}$ is derived from $\boldsymbol{A}_{(i,:)}$ matrix-multiplied by multiple $\boldsymbol{B}$ and $\boldsymbol{C}$ submatrix pairs. Given that there is no data dependency between the computation of $\boldsymbol{A}_{(i,:)}$ and different $\boldsymbol{B}$ and $\boldsymbol{C}$ submatrix pairs, we can use the double buffering technique to achieve overlap between computation and communication. While calculating with submatrices $\boldsymbol{B}_{(:,j)}$ and $\boldsymbol{C}_{(j,:)}$, we can concurrently send them to the buffer of the subsequent rank. To explore the extent of overlap between computation and communication, we calculate the arithmetic intensity of METP as follows:

$$I=\frac{\text{Computation amount}}{\text{Communication volume}}. \tag{4}$$

Based on the matrix multiplication computation formula, the computation performed by our method in each forward pass iteration on each device is $2mk_1k_2/p^2+2mk_2n/p^2$. According to our previous analysis and additionally considering the size of the 16-bit floating point (FP16), the communication volume in each iteration on each device is $\frac{2}{p}(k_1k_2+k_2n)$. Then we have

$$I_{\mathrm{F1}}=\frac{2mk_1k_2/p^2+2mk_2n/p^2}{2(k_1k_2+k_2n)/p}=\frac{m}{p}. \tag{5}$$

The FP16 peak performance of A100 is 312 TFLOPS (trillion floating point operations per second), and the speed of NVLINK is 300 GB/s. Therefore, theoretically, the computation-to-communication bound is equal to

$$\frac{312\ \text{TFLOPS}}{300\ \text{GB/s}}=1040\ \text{FLOPs/byte},$$

where FLOPs means floating point operations.

Therefore, the computation can be well overlapped with communication when we have $m/p>1040$. When considering training a Transformer layer on eight A100 GPUs, setting the batch size to 1 would require the sequence length to be $>1040\times 8=8320$.

According to Algorithm 2, the arithmetic intensity of the backward pass of METP is

$$I_{\mathrm{B1}}=\frac{5mk_1k_2/p^2+4mk_2n/p^2}{4(k_1k_2+k_2n)/p}>I_{\mathrm{F1}}, \tag{6}$$

which means that the forward pass bounds the overlap condition.

## 4 Two-level METP for MHA

MHA consists of four main matrix multiplications and some other element-wise operations, which are

$$(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\boldsymbol{X}\boldsymbol{W}_{\mathrm{qkv}}, \tag{7}$$

$$\boldsymbol{A}=\mathrm{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d}}\right)\boldsymbol{V}, \tag{8}$$

$$\boldsymbol{O}=\boldsymbol{A}\boldsymbol{W}_{\mathrm{proj}}. \tag{9}$$

Eq. (8) has the form of $\boldsymbol{O}=f(\boldsymbol{A}\boldsymbol{B})\boldsymbol{C}$. However, Softmax is not an element-wise operator, so we cannot use Eq. (1) to compute it. Thus, we cannot directly apply Algorithm 1 to it. To serve attention computation, we adapt Algorithms 1 and 2 to create a corresponding version for METP Attention, which involves the online and safe Softmax (Dao et al., 2022) to ensure mathematical equivalence. The details of the forward and backward passes of METP Attention are presented in Algorithms 3 and 4, respectively.

As for the computation of Eqs. (7) and (9), we view $\boldsymbol{A}$ in Eq. (8) as a function $f$ this time. Since

we also want to apply METP to Eq. (8), which has a similar form like $\boldsymbol{O}=f(\boldsymbol{AB})\boldsymbol{C}$, we need to revise the METP algorithm and form a two-level version here.

As depicted in Fig. 2, we propose a two-level METP scheme for MHA, which consists of two loops. The outer loop is responsible for computing $\boldsymbol{Q},\boldsymbol{K}$, and $\boldsymbol{V}$ for different heads and generating partial final results. Meanwhile, the inner loop completes the computation for given heads. The forward and backward passes of METP MHA are detailed in Algorithms 5 and 6, respectively. Similar to TP, we split $\boldsymbol{W}_{\text{qkv}}$ in a column-wise manner and $\boldsymbol{W}_{\text{proj}}$ in a row-wise manner. To ensure that the subsequence on each device uses the same head during attention computation, we use broadcast instead of send/recv to transmit the submatrix $\boldsymbol{W}_{\text{qkv}_i}$ from rank $i$ in the $i^{\text{th}}$ iteration. After each head completes the attention calculation, the results are fed into the projection layer and multiplied with $\boldsymbol{W}_{\text{proj}}$ to produce the partial results of MHA. Broadcasting $\boldsymbol{W}_{\text{proj}}$ is necessary to ensure across-device uniformity. Finally, we compute the results of MHA by aggregating all partial results, each computed with different $\boldsymbol{W}_{\text{qkv}}$ and $\boldsymbol{W}_{\text{proj}}$.

Referring to Li SG et al. (2023), we extend their table describing memory cost and communication cost of MHA of different methods and present our findings in Table 2. We provide a detailed analysis

**Algorithm 3** METP Attention forward pass (inner loop)

Prepare $\boldsymbol{Q}_i$, $\boldsymbol{K}_i^{(0)}$, and $\boldsymbol{V}_i^{(0)}$ for rank $i$
Initialize $\boldsymbol{A}_i$ and $\boldsymbol{L}_i$
Annotate $r_i^j=(i+j)\%p$
**for** $j=0$ to $p-1$ **do**
  $\boldsymbol{E}_{ij}\leftarrow\exp\left(\frac{\boldsymbol{Q}_i\left(\boldsymbol{K}_i^{(j)}\right)^{\mathrm{T}}}{\sqrt{d}}\right)$
  $\boldsymbol{L}_i\leftarrow\boldsymbol{L}_i+\text{rowsum}(\boldsymbol{E}_{ij})$
  $\boldsymbol{A}_i\leftarrow\boldsymbol{A}_i+\boldsymbol{E}_{ij}\boldsymbol{V}_i^{(j)}$
  Send $\boldsymbol{K}_i^{(j)}$ and $\boldsymbol{V}_i^{(j)}$ to rank $r_i^{j-1}$
  Receive $\boldsymbol{V}_{i+1}^{(j)}$ and $\boldsymbol{K}_{i+1}^{(j)}$ from rank $r_i^{j+1}$
  $\boldsymbol{V}_i^{(j+1)}\leftarrow\boldsymbol{V}_{i+1}^{(j)}$, $\boldsymbol{K}_i^{(j+1)}\leftarrow\boldsymbol{K}_{i+1}^{(j)}$
**end for**
$\boldsymbol{A}_i\leftarrow\boldsymbol{A}_i\odot\boldsymbol{L}_i^{-1}$
Return $\boldsymbol{A}_i$

**Algorithm 4** METP Attention backward pass (inner loop)

Prepare $\boldsymbol{Q}_i$, $\boldsymbol{K}_i^{(0)}$, $\boldsymbol{V}_i^{(0)}\in\mathbb{R}^{b\times(s/p)\times(h/p)}$, $\Delta\boldsymbol{A}_i$, $\boldsymbol{A}_i\in\mathbb{R}^{b\times(s/p)\times h}$, and $\boldsymbol{L}_i\in\mathbb{R}^{b\times(s/p)}$ for rank $i$
Initialize $\Delta\boldsymbol{Q}_i$, $\Delta\boldsymbol{K}_i^{(0)}$, and $\Delta\boldsymbol{V}_i^{(0)}$
Annotate $r_i^j=(i+j)\%p$
$\boldsymbol{D}_i\leftarrow\text{rowsum}(\Delta\boldsymbol{A}_i\odot\boldsymbol{A}_i)$
**for** $j=0$ to $p-1$ **do**
  $\boldsymbol{S}_{ij}\leftarrow\frac{\boldsymbol{Q}_i\left(\boldsymbol{K}_i^{(j)}\right)^{\mathrm{T}}}{\sqrt{d}}$
  $\boldsymbol{A}_{ij}\leftarrow\exp(\boldsymbol{S}_{ij})\odot\boldsymbol{L}_i^{-1}$
  $\Delta\boldsymbol{A}_{ij}\leftarrow\Delta\boldsymbol{A}_i\left(\boldsymbol{V}_i^{(j)}\right)^{\mathrm{T}}$
  $\Delta\boldsymbol{V}_i^{(j)}\leftarrow\Delta\boldsymbol{V}_i^{(j)}+\boldsymbol{A}_{ij}^{\mathrm{T}}\Delta\boldsymbol{A}_i$
  $\Delta\boldsymbol{S}_{ij}\leftarrow\boldsymbol{A}_{ij}\odot(\Delta\boldsymbol{A}_{ij}-\boldsymbol{D}_i)$
  $\Delta\boldsymbol{Q}_i\leftarrow\Delta\boldsymbol{Q}_i+\Delta\boldsymbol{S}_{ij}\boldsymbol{K}_i^{(j)}$
  $\Delta\boldsymbol{K}_i^{(j)}\leftarrow\Delta\boldsymbol{K}_i^{(j)}+\Delta\boldsymbol{S}_{ij}^{\mathrm{T}}\boldsymbol{Q}_i$
  Send $\boldsymbol{K}_i^{(j)}$, $\boldsymbol{V}_i^{(j)}$, $\Delta\boldsymbol{V}_i^{(j)}$, and $\Delta\boldsymbol{K}_i^{(j)}$ to rank $r_i^{j-1}$
  Receive $\boldsymbol{V}_{i+1}^{(j)}$, $\boldsymbol{K}_{i+1}^{(j)}$, $\Delta\boldsymbol{V}_{i+1}^{(j)}$, and $\Delta\boldsymbol{K}_{i+1}^{(j)}$ from rank $r_i^{j+1}$
  $\boldsymbol{V}_i^{(j+1)}\leftarrow\boldsymbol{V}_{i+1}^{(j)}$, $\boldsymbol{K}_i^{(j+1)}\leftarrow\boldsymbol{K}_{i+1}^{(j)}$
  $\Delta\boldsymbol{V}_i^{(j+1)}\leftarrow\Delta\boldsymbol{V}_{i+1}^{(j)}$, $\Delta\boldsymbol{K}_i^{(j+1)}\leftarrow\Delta\boldsymbol{K}_{i+1}^{(j)}$
**end for**
Return $\Delta\boldsymbol{Q}_i$, $\Delta\boldsymbol{K}_i$, and $\Delta\boldsymbol{V}_i$

**Algorithm 5** METP MHA forward pass (outer loop)

Prepare $\boldsymbol{X}_i\in\mathbb{R}^{b\times(s/p)\times h}$, $\boldsymbol{W}_{\text{proj}_i}\in\mathbb{R}^{h\times(3h/p)}$, and $\boldsymbol{W}_{\text{proj}_i}\in\mathbb{R}^{(h/p)\times h}$ for rank $i$
Initialize $\boldsymbol{O}_i\in\mathbb{R}^{b\times(s/p)\times h}$
**for** $j=0$ to $p-1$ **do**
  From rank $j$, broadcast $\boldsymbol{W}_{\text{qkv}_j}$ and $\boldsymbol{W}_{\text{proj}_j}$
  $(\boldsymbol{Q}_{ij},\boldsymbol{K}_{ij},\boldsymbol{V}_{ij})\leftarrow\boldsymbol{X}_i\boldsymbol{W}_{\text{qkv}_j}$
  $\boldsymbol{A}_{ij}\leftarrow\text{METP\_Attention}(\boldsymbol{Q}_{ij},\boldsymbol{K}_{ij},\boldsymbol{V}_{ij})$
  $\boldsymbol{O}_i\leftarrow\boldsymbol{O}_i+\boldsymbol{A}_{ij}\boldsymbol{W}_{\text{proj}_j}$
**end for**
Return $\boldsymbol{O}_i$

**Algorithm 6** METP MHA backward pass (outer loop)

Prepare $\boldsymbol{X}_i$, $\Delta\boldsymbol{O}_i$, $\boldsymbol{A}_i\in\mathbb{R}^{b\times(s/p)\times h}$, $\boldsymbol{W}_{\text{qkv}_i}\in\mathbb{R}^{h\times(3h/p)}$, $\boldsymbol{W}_{\text{proj}_i}\in\mathbb{R}^{(h/p)\times h}$, and $\boldsymbol{L}_i\in\mathbb{R}^{b\times(s/p)}$ for rank $i$
Initialize $\Delta\boldsymbol{X}_i\in\mathbb{R}^{b\times(s/p)\times h}$
**for** $j=0$ to $p-1$ **do**
  From rank $j$, broadcast $\boldsymbol{W}_{\text{qkv}_j}$ and $\boldsymbol{W}_{\text{proj}_j}$
  $\Delta\boldsymbol{W}_{\text{proj}_j}\leftarrow\boldsymbol{A}_{ij}^{\mathrm{T}}\Delta\boldsymbol{O}_i$
  $(\boldsymbol{Q}_{ij},\boldsymbol{K}_{ij},\boldsymbol{V}_{ij})\leftarrow\boldsymbol{X}_i\boldsymbol{W}_{\text{qkv}_j}$
  $\Delta\boldsymbol{A}_{ij}\leftarrow\Delta\boldsymbol{O}_i\boldsymbol{W}_{\text{proj}_j}^{\mathrm{T}}$
  $(\Delta\boldsymbol{Q}_{ij},\Delta\boldsymbol{K}_{ij},\Delta\boldsymbol{V}_{ij})\leftarrow\text{METP\_Attention\_Backward}(\boldsymbol{Q}_{ij},\boldsymbol{K}_{ij},\boldsymbol{V}_{ij},\boldsymbol{A}_{ij},\Delta\boldsymbol{A}_{ij},\boldsymbol{L}_i)$
  $\Delta\boldsymbol{X}_i\leftarrow\Delta\boldsymbol{X}_i+[\Delta\boldsymbol{Q}_{ij},\Delta\boldsymbol{K}_{ij},\Delta\boldsymbol{V}_{ij}]\boldsymbol{W}_{\text{proj}_j}^{\mathrm{T}}$
  $\Delta\boldsymbol{W}_{\text{proj}_j}=\boldsymbol{X}_i^{\mathrm{T}}[\Delta\boldsymbol{Q}_{ij},\Delta\boldsymbol{K}_{ij},\Delta\boldsymbol{V}_{ij}]$
  Reduce $\Delta\boldsymbol{W}_{\text{qkv}_j}$ and $\Delta\boldsymbol{W}_{\text{proj}_j}$ to rank $j$
**end for**
Return $\Delta\boldsymbol{X}_i$, $\Delta\boldsymbol{W}_{\text{qkv}_i}$, and $\Delta\boldsymbol{W}_{\text{proj}_i}$

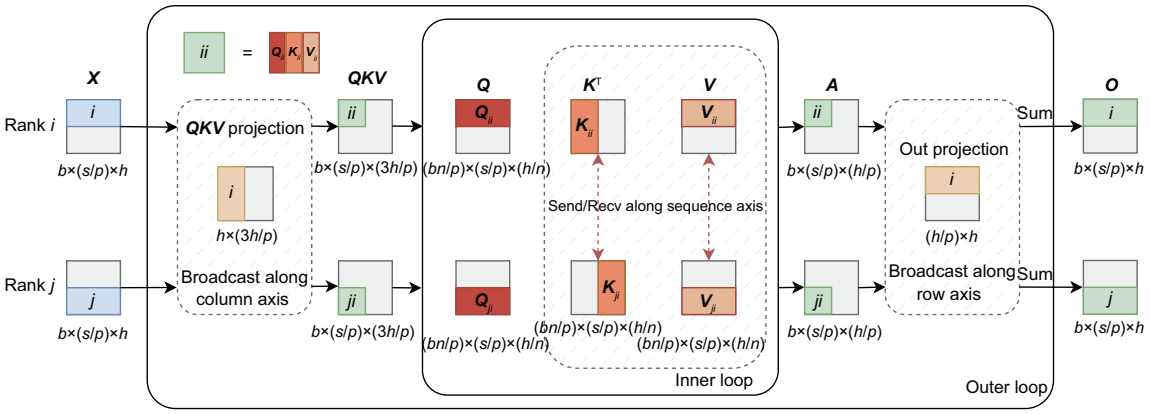


**Fig. 2 METP for MHA ($b$: batch size; $s$: sequence length; $h$: hidden size; $p$: parallel degree; $n$: number of heads. Note: $h/n$=head_size. $QKV$ will be transposed, reshaped, and split to obtain $Q, K, V$ before they are fed into the inner loop. The workflow of the inner loop is similar to Fig. 1e)**

of the results below.

### 4.1 Memory consumption

In this subsection, we analyze the memory consumption of our method in training MHA layers.

#### 4.1.1 Model states

Each MHA layer consists of two projection weight matrices: a QKV projection matrix with a shape of $h \times 3h$ and an attention-out-projection matrix with a shape of $h \times h$. Therefore, the total memory consumption for these weight matrices is $4h^2 + 4h$, where the additional $4h$ represents the number of bias parameters. Additionally, for each parameter matrix, we need to store its gradient matrix and corresponding optimizer states. The Adam optimizer holds momentum and variance for each parameter, costing 2× memory cost of the parameter. Thus, using the Adam optimizer requires 4× memory consumption of weight matrices. We evenly distribute the model parameters across all devices in the communication group. Finally, each Transformer layer's model state memory consumption is $(16h^2 + 16h)/p$ on each device.

#### 4.1.2 Intermediate results

According to Algorithm 6, we store the input $\boldsymbol{X}$ in Eq. (7) and the attention results of Eq. (8) for backward propagation, resulting in a memory cost of $2bsh/p$.

During the computation of MHA without FlashAttention, the maximum memory cost happens when computing attention scores. We need to prepare $\boldsymbol{Q}_{ij}$, $\boldsymbol{K}_{ij}$, and $\boldsymbol{V}_{ij}$ for each inner loop. Note that the sequence lengths of $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ are the same during training. Since the shapes of matrices $\boldsymbol{Q}_{ij}$, $\boldsymbol{K}_{ij}$, and $\boldsymbol{V}_{ij}$ are $b \times n/p \times s/p \times h/n$, the memory cost of temporarily storing them is $3b(s/p)(n/p)(h/n) = 3bsh/p^2$. To compute the subresult of attention scores $\boldsymbol{E}_{ij}$ in line 5 of Algorithm 3, whose shape is $b \times n/p \times s/p \times s/p$, it requires additional $bns^2/p^3$ memory. Finally, the intermediate result consumption of METP MHA is $2bsh/p + 3bsh/p^2 + bns^2/p^3$. Accumulatively, we obtain the result recorded in Table 2. Note that we omit the small overheads like bias and rowsum. When $s >> h$, the memory consumption of our method is $O(s^2/p^3)$, whereas those of the others are $O(s^2/p)$. This shows a $p^2$ memory optimization compared with other methods. FlashAttention can reduce the memory consumption of attention scores because it uses shared memory instead of HBM for computing attention scores. With FlashAttention, the peak memory overhead of METP's intermediate results is $2bsh/p + 3bsh/p^2$. When comparing the peak memory of intermediate results with TP, we have

$$r = \frac{\frac{3bsh}{p^2} + \frac{2bsh}{p}}{\frac{4bsh}{p} + bsh} = \frac{3 + 2p}{p(4 + p)}, \tag{10}$$

where $r$ is the ratio of peak memory of METP to TP. It means that the peak memory consumption of a single layer's intermediate results of METP is at least 41.7% ($p$=2) less than that of TP. As for the intermediate result memory consumption after forward propagation, we can easily obtain that METP saves

**Table 2 Memory overhead and communication volume of an MHA layer during training**

| Method | Operation | Shape of matrix 1 | Shape of matrix 2 | Shape of output |
|---|---|---|---|---|
| Megatron-LM (TP+SP) | AG+$\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ | $b\times(s/p)\times h$ | $h\times(h/p)$ | $b\times(n/p)\times s\times(h/n)$ |
| | $\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}$ | $b\times(n/p)\times s\times(h/n)$ | $b\times(n/p)\times s\times(h/n)$ | $b\times(n/p)\times s\times s$ |
| | $\boldsymbol{A}\boldsymbol{V}$ | $b\times(n/p)\times s\times s$ | $b\times(n/p)\times s\times(h/n)$ | $b\times(n/p)\times s\times(h/n)$ |
| | Linear+RS | $b\times(n/p)\times s\times(h/n)$ | $(h/p)\times h$ | $b\times(s/p)\times h$ |
| RSA | $\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ | $b\times(s/p)\times h$ | $h\times h$ | $b\times n\times(s/p)\times(h/n)$ |
| | Ring-$\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}$ | $b\times n\times(s/p)\times(h/n)$ | $b\times n\times(s/p)\times(h/n)$ | $b\times n\times(s/p)\times s$ |
| | Ring-$\boldsymbol{A}\boldsymbol{V}$ | $b\times n\times(s/p)\times s$ | $b\times n\times(s/p)\times(h/n)$ | $b\times n\times(s/p)\times(h/n)$ |
| | Linear | $b\times n\times(s/p)\times(h/n)$ | $h\times h$ | $b\times(s/p)\times h$ |
| FSDP/ZeRO+Ulysses | AG+$\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ | $b\times(s/p)\times h$ | $(h/p)\times h$ | $b\times n\times(s/p)\times(h/n)$ |
| | A2A+$\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}$ | $b\times n\times(s/p)\times(h/n)$ | $b\times n\times(s/p)\times(h/n)$ | $b\times(n/p)\times s\times s$ |
| | $\boldsymbol{A}\boldsymbol{V}$+A2A | $b\times(n/p)\times s\times s$ | $b\times(n/p)\times s\times(h/n)$ | $b\times n\times(s/p)\times(h/n)$ |
| | AG+Linear | $b\times n\times(s/p)\times(h/n)$ | $(h/p)\times h$ | $b\times(s/p)\times h$ |
| METP (ours) | Broadcast-$\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ | $b\times(s/p)\times h$ | $h\times(h/p)$ | $b\times(n/p)\times(s/p)\times(h/n)$ |
| | P2P-$\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}$ | $b\times(n/p)\times(s/p)\times(h/n)$ | $b\times(n/p)\times(s/p)\times(h/n)$ | $b\times(n/p)\times(s/p)\times(s/p)$ |
| | P2P-$\boldsymbol{A}\boldsymbol{V}$ | $b\times(n/p)\times(s/p)\times(s/p)$ | $b\times(n/p)\times(s/p)\times(h/n)$ | $b\times(n/p)\times(s/p)\times(h/n)$ |
| | Broadcast-Linear | $b\times(n/p)\times(s/p)\times(h/n)$ | $(h/p)\times h$ | $b\times(s/p)\times h$ |

| Method | Memory | Peak memory | Communication volume |
|---|---|---|---|
| Megatron-LM (TP+SP) | $\frac{16h^2}{p}+\frac{5bsh}{p}$ | $\frac{16h^2}{p}+\frac{4bsh}{p}+\frac{bns^2}{p}k+bsh$ | $5\frac{p-1}{p}bsh$ |
| RSA | $16h^2+\frac{5bsh}{p}$ | $16h^2+\frac{5bsh}{p}+\frac{bns^2}{p}$ | $8\frac{p-1}{p}bsh$ |
| FSDP/ZeRO+Ulysses | $\frac{16h^2}{p}+\frac{5bsh}{p}$ | $3h^2+\frac{13h^2}{p}+\frac{bns^2}{p}k+\frac{5bsh}{p}$ | $12\frac{p-1}{p}h^2+8\frac{p-1}{p^2}bsh$ |
| METP (ours) | $\frac{16h^2}{p}+\frac{2bsh}{p}$ | $\frac{16h^2}{p}+\frac{3bsh}{p^2}+\frac{bns^2}{p^3}k+\frac{2bsh}{p}$ | $12(\log_2 p)h^2+6\frac{p-1}{p}bsh$ |

$b$: batch size; $s$: sequence length; $h$: hidden size; $p$: parallel degree; $n$: number of heads. If flash attention is used, $k=0$; otherwise, $k=1$. Memory: the memory overhead after executing forward propagation. Peak memory: the peak memory overhead during forward propagation. Note: we ignore the reshape and transpose operations in this table. For example, the result of AG+$\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ in TP should be first reshaped to $b\times s\times(n/p)\times(h/n)$, and then transposed to $b\times(n/p)\times s\times(h/n)$. AG: AllGather; RS: ReduceScatter; A2A: AllToAll; P2P: peer-to-peer

60% of memory. As $p$ enlarges, METP could save more peak memory. When the number of layers is $L$, the actual ratio becomes

$$\bar{r}=\frac{r}{L}+\frac{2}{5}\frac{L-1}{L}. \tag{11}$$

Thus, we can save at least 41.7% of the memory when computing MHA.

## 4.2 Communication overhead

### 4.2.1 Forward pass

During the outer loop of the computation of MHA, the $\boldsymbol{QKV}$ projection needs to broadcast a matrix of dimension $h\times(3h/p)$. This results in a communication volume of $3(\log_2 p)h^2/p$. Repeating this process over $p$ iterations, we obtain an overall communication volume of $3(\log_2 p)h^2$. Similarly, the total communication volume for the out projection is $(\log_2 p)h^2$.

The inner-loop computation follows the communication style consistent with METP. Each iteration results in the communication volume of $2bsh/p^2$ for $\boldsymbol{K}$ and $\boldsymbol{V}$. According to Section 3.2.1, we need only to do $p-1$ times communication, resulting in $2\frac{p-1}{p^2}bsh$ communication volume for each outer loop. Since outer loops iterate $p$ times, each device has an accumulated communication volume of $2\frac{p-1}{p}bsh$ for $\boldsymbol{K}$ and $\boldsymbol{V}$.

Thus, the overall communication volume of the forward pass is $4(\log_2 p)h^2+2\frac{p-1}{p}bsh$.

### 4.2.2 Backward pass

In the backward computation process of MHA, according to Algorithm 6, we need to compute the gradients for the output of METP Attention and the input of MHA, which requires broadcasting the $\boldsymbol{QKV}$ projection and out projection weight matrices, resulting in a communication volume of $4(\log_2 p)h^2$. Additionally, to compute the gradients of each submatrix of weights, we use the Reduce operation to gather the gradients from each rank, which incurs another $4(\log_2 p)h^2$ communication volume. Similarly, in the computation of gradients for the input of METP Attention, communication

of submatrices of $\boldsymbol{K}$ and $\boldsymbol{V}$ is required in the inner loop, resulting in a communication volume of $2\frac{p-1}{p}bsh$. Furthermore, as shown in Algorithm 4, during the gradient calculation process for $\boldsymbol{K}$ and $\boldsymbol{V}$, the transmission of gradients $\Delta\boldsymbol{K}$ and $\Delta\boldsymbol{V}$ is necessary. Each iteration generates a communication volume of $2bsh/p^2$, resulting in a total communication volume of $2\frac{p-1}{p}bsh$. Finally, the overall communication volume of the backward pass of METP MHA is $8(\log_2 p)h^2+4\frac{p-1}{p}bsh$.

Table 2 summarizes the communication volume compared with other methods.

### 4.3 Overlap between computation and communication

Fig. 3 shows the overlap execution of an METP MHA layer with $p=4$. Before each outer loop iteration $i$, we launch a broadcast to prepare the corresponding $\boldsymbol{W}_{\mathrm{QKV}_i}$ and $\boldsymbol{W}_{\mathrm{out}i}$ for it. In the inner loop, which is performed by Algorithm 3, we overlap the computation with P2P send and receive operations. Similarly, we can overlap the Broadcast and Reduce operations of the $(i+1)^{\mathrm{th}}$ iteration with the calculation of the $i^{\mathrm{th}}$ iteration.

To illustrate the possibility of overlap, we compute the arithmetic intensity of the METP MHA forward pass.

#### 4.3.1 Outer loop

The computation amount of Eqs. (7)–(9) is approximately $6bsh^2$, $4bs^2h$, and $2bsh^2$, respectively. The total amount is about $8bsh^2/p+4bs^2h/p$. The communication volume of FP16 forward propagation of each outer loop iteration is $2(4(\log_2 p)h^2/p+\frac{p-1}{p^2}2bsh)$. Thus, the arithmetic intensity is

$$I_{\mathrm{F2}}=\frac{8bsh^2+4bs^2h}{(\log_2 p)8h^2+\frac{p-1}{p}4bsh}>\frac{8bsh+4bs^2}{(\log_2 p)8h+4bs}. \tag{12}$$

Supposing that we have $p=8$, $b=1$, we can simplify the inequality as follows:

$$I_{\mathrm{F2}}>\frac{4s(2h+s)}{24h+4s}>\frac{4s(2h+s)}{12(2h+s)}=\frac{s}{3}. \tag{13}$$

When $\frac{s}{3}>1040$, i.e., $s>3120$, the computation and communication can be theoretically fully overlapped within an A100 server.

For the backward pass, the communication volume is twice that in the forward pass, and the computation amount is more than twice due to recomputation. Thus, the arithmetic intensity of the backward pass is greater than that of the forward pass. Therefore, the condition is bounded by the forward pass.

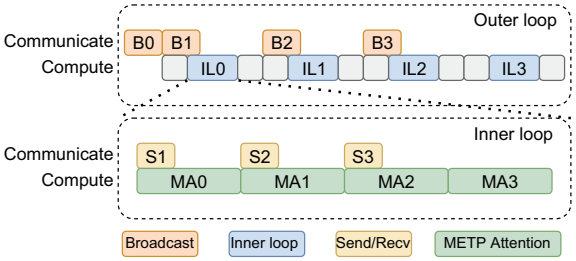


**Fig. 3 Overlap between communication and computation in METP MHA**

#### 4.3.2 Inner loop

Since we form the inner loop like Algorithm 3, we follow the condition in Section 3.3. Finally, for METP MHA, we require a sequence length to be >8320 for full overlap.

## 5 Evaluation

### 5.1 Experimental setup

1. Environments

We conducted experiments on a machine with eight NVIDIA A100-SXM4-80 GB GPUs interconnected through NVLink. The CPU is an Intel® Xeon® Platinum 8369B CPU @2.90 GHz. The software environment includes PyTorch 2.0, compute unified device architecture (CUDA) 11.8, and flash_attn 2.2.5.

2. Implementation

We implemented two versions of the METP Transformer using PyTorch and flash_attn library. We implemented the attention calculation in the first version using PyTorch Aten operators. The second version used the FlashAttention-2 kernel because FlashAttention-3 is compatible only with the Hopper architecture, and we conducted our experiments on the Ampere architecture. We carefully managed the online Softmax outside the kernel to make these two implementations mathematically equivalent. The first version was used in environments that do not support FlashAttention. Since A100 supports FlashAttention-2, our evaluations below are primarily based on the second version.

3. Models

We conducted experiments to evaluate the performance of METP in training large-scale models with a large number of parameters, including BERT-Large (Devlin et al., 2019), LLaMA-7B, LLaMA-70B (Touvron et al., 2023), and GPT-175B (Brown et al., 2020). The hyper-parameters of our chosen models are shown in Table 3. We examined all the four model types' memory consumption and the maximum sequence length that they can reach. Given that the entire LLaMA-70B (Touvron et al., 2023) and GPT-175B (Brown et al., 2020) cannot fit in one server with eight cards and are required to train with PP, we tested the throughput for one PP stage (Narayanan et al., 2021a) and ignored the pipeline bubbles that occurred in multiple stages. This means that we evaluated only a few layers of LLaMA-70B (Touvron et al., 2023) and GPT-175B (Brown et al., 2020).

4. METP setting

We applied the original METP from Section 3 for the Transformer's FFN layer and the two-level METP for MHA from Section 4, following the partition schemes listed in Tables 1 and 2.

5. Comparison methods

We chose TP implemented by Megatron-LM and Ulysses (Jacobs et al., 2024) implemented by DeepSpeed (Rasley et al., 2020) as our comparison methods. TP was combined with SP (Korthikanti et al., 2023) in our experiments, while Ulysses was used with ZeRO from DeepSpeed. By applying FlashAttention, the memory overhead of training Transformers increased linearly with the increase of sequence length. Since RSA-SP is inherently incompatible with FlashAttention, it quickly exceeded the memory limit as the sequence length grew. Although we have analyzed this in the previous sections, we did not include it in our comparisons. We compared these three methods in terms of memory consumption and model FLOPS utilization (MFU) (Chowdhery et al., 2022) under various settings.

### 5.2 Correctness evaluation

To check whether extra communications affect the precision, we compared our implementation with the naive implementation of TransformerEncoderLayer in PyTorch. Then, by inputting different inputs and comparing the output and gradient results of forward and backward propagation, we verified the correctness of our implementation. Moreover, we tested the METP MHA function and METP FFN function individually. Our experimental results showed that the average error of each element was $<10^{-5}$, which means that our implementation is mathematically equivalent to the naive implementation in PyTorch.

### 5.3 Memory efficiency evaluation

To evaluate the memory efficiency of METP, we measured the memory consumption using different methods, using TP as the baseline. We calculated the ratios of memory consumption for each method relative to TP. As illustrated in Fig. 4, the ratio of METP to TP decreased as the sequence length increased, indicating that METP becomes more efficient than TP with longer sequences. For example, in GPT-175B training, METP reduced the memory consumption for training long sequences by up to 64% (i.e., in the 512k case). This significant reduction highlights its potential for larger batch sizes and larger sequence lengths.

### 5.4 MFU evaluation

We investigated the MFU of each setting, which is proportional to throughput. MFU is computed via

$$\mathrm{MFU}=\frac{\text{Model FLOPs per iteration}}{\text{iter\_time}\times\text{Hardware FLOPS}}, \tag{14}$$

which indicates the efficiency of using computing resources.

**Table 3 Model configuration**

| Model | Hidden size | Number of heads | Number of layers | Number of layers per device |
|---|---|---|---|---|
| BERT-Large | 1024 | 16 | 24 | 24 |
| LLaMA-7B | 4096 | 32 | 32 | 32 |
| LLaMA-70B | 8192 | 64 | 80 | 5 |
| GPT-175B | 12 288 | 96 | 96 | 1 |

LLaMA configurations are from https://github.com/aws-neuron/neuronx-nemo-megatron; GPT-175B configuration is from Narayanan et al. (2021a)

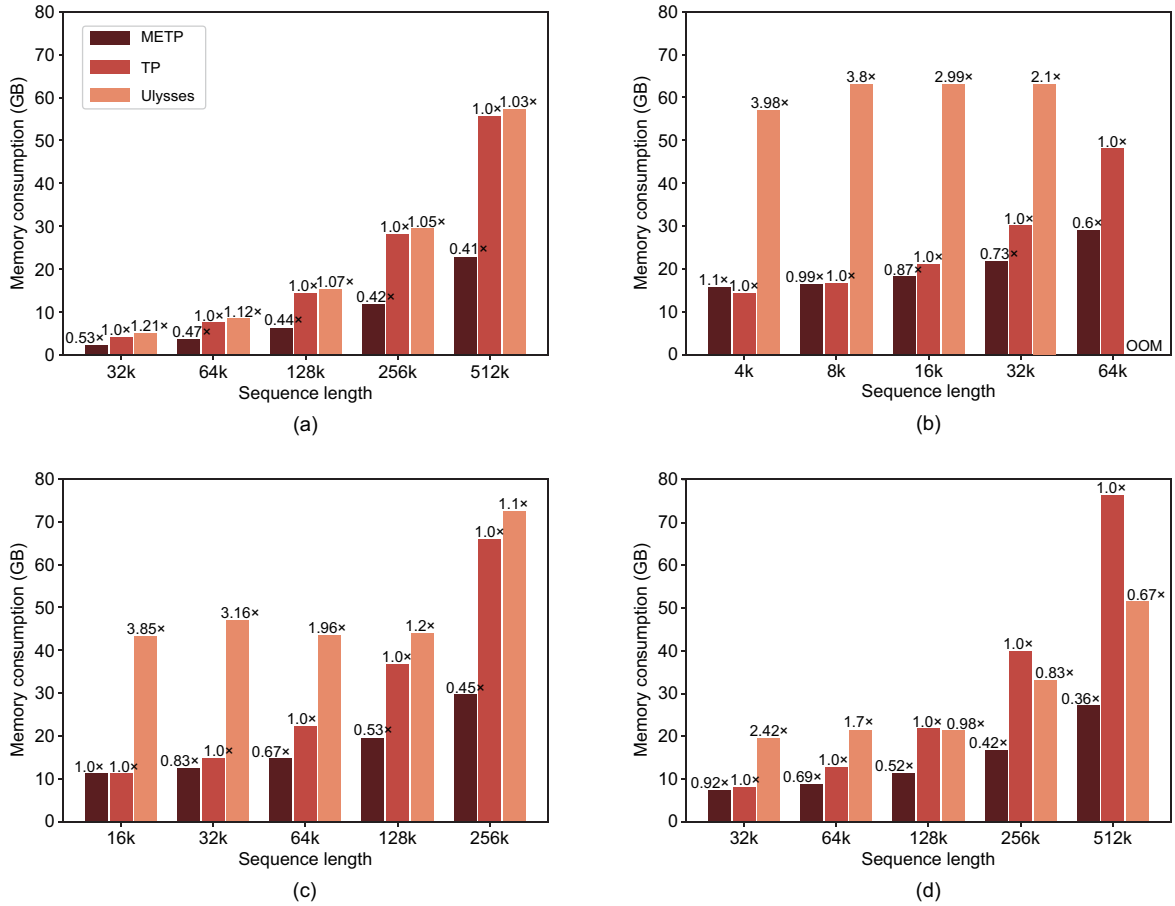


**Fig. 4 Memory usage of different methods at different sequence lengths on eight A100 GPUs: (a) BERT-Large; (b) LLaMA-7B; (c) LLaMA-70B; (d) GPT-175B. Above each bar, we give their ratios compared to TP. OOM: out-of-memory**

Simultaneously, we set the batch size for each configuration to its maximum value without exceeding the memory limit, as larger batch sizes can more effectively leverage GPU resources. Given that METP is designed for long-sequence training, we limited the sequence length to at least 32k in this subsection. Fig. 5 illustrates the MFU and corresponding batch sizes for various models trained with different methods across multiple sequence lengths. Our results showed that the MFU of METP outperformed that of TP in most cases and that METP can train the model with longer sequences than TP and Ulysses. METP outperformed TP because TP has communication that cannot be overlapped. Ulysses outperformed METP in BERT-Large, LLaMA-70B, and GPT-175B because it caches the model parameters, avoiding heavy parameter AllGather communication in ZeRO. In these cases, Ulysses performed much less communication than TP and METP. However, once the model slices were too big on a device, as shown in Fig. 5b, Ulysses may suffer from performance degradation.

### 5.5 Scalability evaluation

In this subsection, we measured the scalability of different methods by accounting for the maximum sequence lengths under various degrees of parallelism. For each kind of model, we first measured the maximum sequence length they can reach on a single device during training and used it to normalize the results in the scaling experiment. LLaMA-7B was not included in this subsection because it encountered an out-of-memory (OOM) problem in the single device case. Fig. 6 shows each method's normalized maximum sequence length on different parallel degrees. The results showed that TP and

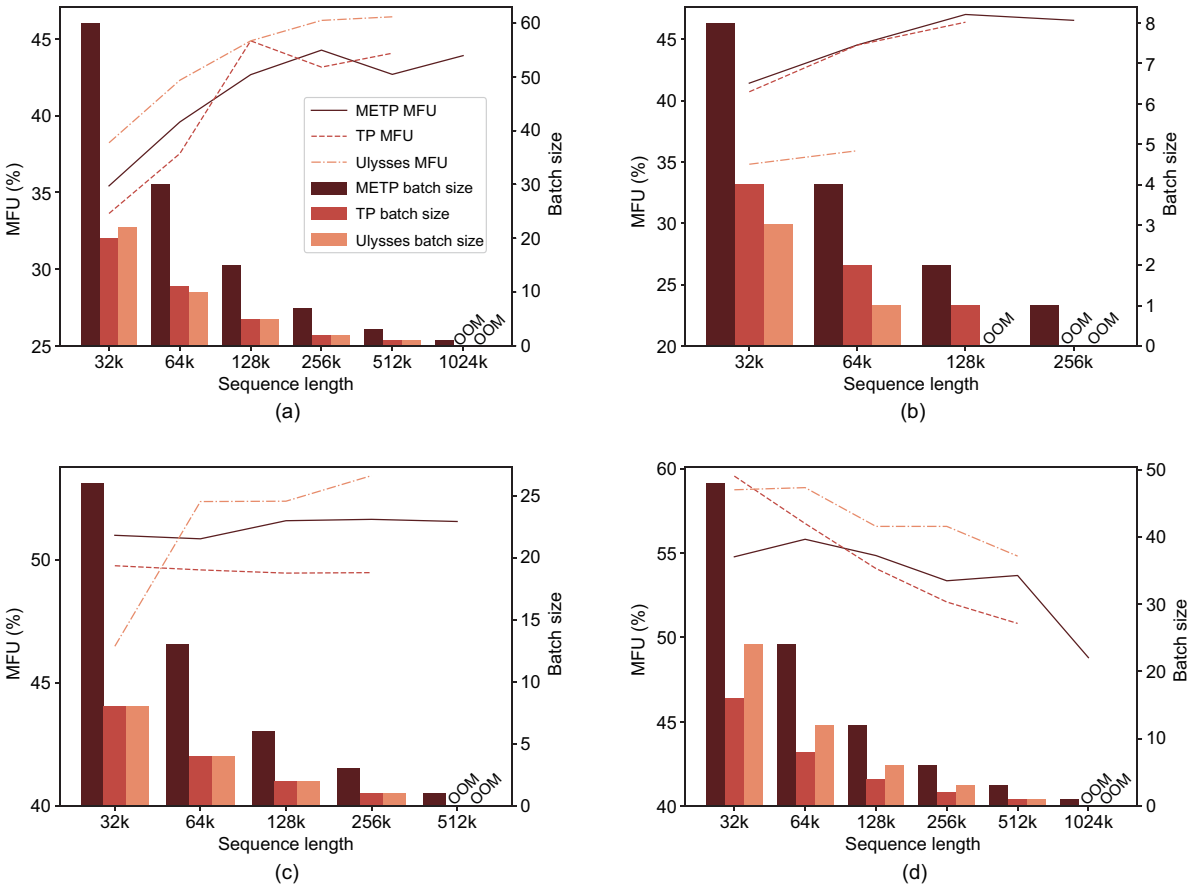


**Fig. 5 MFU and the corresponding batch size for different models: (a) BERT-Large; (b) LLaMA-7B; (c) LLaMA-70B; (d) GPT-175B. OOM: out-of-memory**

Ulysses followed a linear scalability while METP had a superlinear scalability. According to our memory consumption analysis in Tables 1 and 2, METP reduced the overhead of most activation values to $O(1/p^2)$, whereas the overheads of TP and Ulysses were still $O(1/p)$. This can explain why METP has better and superlinear scalability than other methods.

LLaMA-70B consumed a significant amount of model state memory, especially when the number of devices was limited. As the number of devices increased, the memory overheads were distributed across each rank, allowing more space for each rank to store sequences. Consequently, the normalized values were much larger than those of BERT-Large and GPT-175B.

Given this good scalability, METP achieved a more considerable maximum sequence length (increased by 2.38× to 2.99×) compared to other methods among all models on eight A100 GPUs.

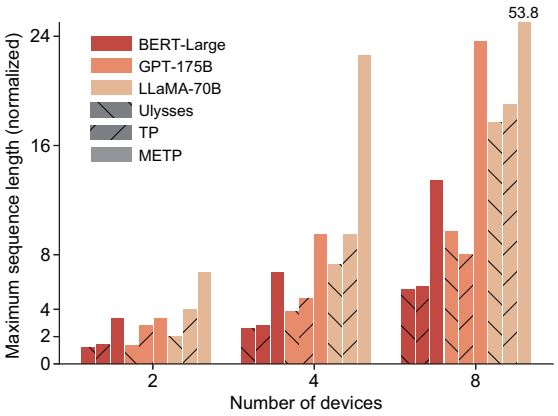


**Fig. 6 Scaling sequence length by increasing the number of devices. Results are normalized by the maximum sequence length that a single device can achieve for each model setting**

## 5.6 Overlap evaluation

We investigated the gain and condition of overlapping by running METP synchronously and asynchronously. In Fig. 7, "Comp" and "Comm" represent

the ratios of computation and communication time in a synchronous setting. "Async" is the ratio of the asynchronous setting compared to the synchronous setting when applying a double buffering technique for overlapping. Fig. 7 shows that the asynchronous setting outperformed the synchronous setting in all our experiments. Additionally, as the sequence length increased, the difference between "Async" and "Comp" became smaller.

## 6 Discussion

### 6.1 Interpretation of results

Fig. 4 illustrates that METP exhibits slightly higher memory consumption than TP in the 4k sequence length case for LLaMA-7B. This is attributed to METP maintaining a double buffer for communication, which adds to the memory overhead. Additionally, Ulysses appears less memory-efficient in short-sequence scenarios because ZeRO gathers and caches all model parameters, leading to higher memory usage. However, as the sequence length increases, the primary memory consumption shifts to activations, and Ulysses becomes more memory-efficient than TP.

Fig. 5 shows that in short-sequence cases, METP is slightly worse than TP and Ulysses. One reason is that METP recomputes $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ during running Algorithm 6, whereas TP and Ulysses do not, which brings additional computation overheads. Another reason is that METP splits the computation task into $p^3$ tiles, which might lead to poor streaming multiprocessor (SM) efficiency when the tiles are too small. We claim that METP is designed for long-sequence training, which is acceptable to us.

Based on our prior analysis, the sequence length must exceed 8320 to achieve overlap between computation and communication. Our overlap experiments indicate that once the sequence length surpasses 8k, the overlap performance essentially stabilizes. However, this does not work for Fig. 7a. This is because our analysis related to the computation-to-communication bound is idealized. Memory-bound element-wise operators may reduce overall computational performance in the real environment.

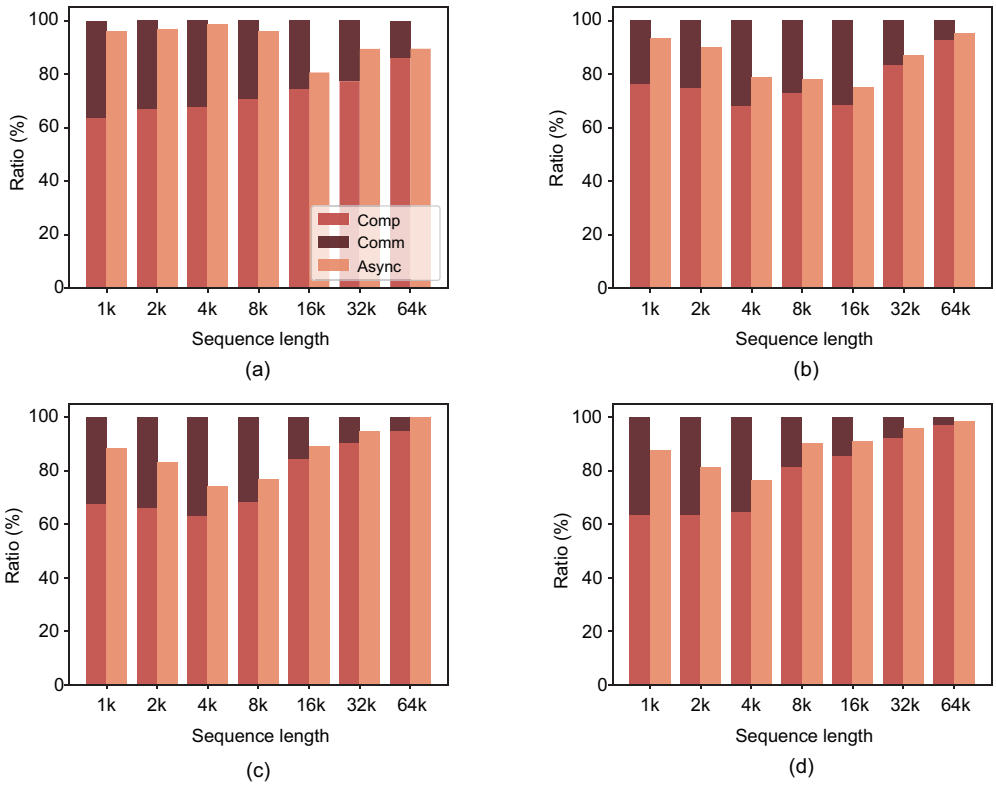


**Fig. 7 Ratios of computation and communication of METP: (a) BERT-Large; (b) LLaMA-7B; (c) LLaMA-70B; (d) GPT-175B**

Additionally, when the sequence length is small, the GPU is hard to be fully used with limited threads, resulting in poor performance. Consequently, we cannot simply use the peak device performance to determine the critical threshold of overlap. Moreover, the communication via NVIDIA Collective Communication Library (NCCL) follows the alpha-beta model (Li A et al., 2020). When the data volume is too small, the actual bandwidth decreases sharply. For instance, according to our testing, the bandwidth for sending and receiving 1 MB of data is only 37 GB/s. Additionally, our current implementation considers only the overlap within a layer, where the first Broadcast and the last Reduce operations cannot overlap with computation, resulting in a subtly more significant difference between "Async" and "Comp."

### 6.2 Integration with other methods

METP is orthogonal to DP, FSDP, and PP. Thus, we could combine METP with them to train LLMs without modification. We now explain why they are orthogonal. METP partitions tensors along different axes from DP, making them naturally orthogonal. Although both FSDP and METP split parameters, they operate at different stages of layer execution. Specifically, FSDP prepares parameters before a layer's execution begins, while METP is integrated directly within the layer's execution process. Therefore, FSDP and METP are orthogonal. PP splits the model into multiple stages, and METP is used within the stages. Therefore, PP and METP are orthogonal.

METP can also work with TP with a few modifications. We can treat the partitioned parameter matrices as non-partitioned ones for METP and apply METP to them. After the execution of METP, we can finish the AllReduce operation of TP to finish the execution.

### 6.3 Possible scenarios to use METP

1. Hybrid parallelism. METP generates higher communication volume to achieve lower memory consumption, making it particularly suitable for deployment on high-speed interconnects. Recent new clusters, such as DGX GB200 NVL72 (https://www.nvidia.com/en-us/data-center/gb200-nvl72/) and DGX H100 SuperPOD (https://docs.nvidia.com/https:/docs.nvidia.com/dgx-superpod-reference-architecture-dgx-h100.pdf), apply NVLink switch that supports high-speed and simultaneous communication among dozens of GPUs. This allows users to apply hybrid parallelism methods (Lai et al., 2023) consisting of more intra-operator parallelism methods to avoid bubble and load-balance problems in inter-operator parallelism during training. Applying METP to these servers can achieve lower memory usage while possibly performing better.

2. Long-sequence scenarios. In addition to language models, recent applications like video generation (Liu YX et al., 2024) rely on long-sequence data. For instance, generating a 2-s video of 720p resolution can produce one million tokens according to the setting of Open-Sora (https://github.com/hpcaitech/Open-Sora).

3. Limited memory cluster. In clusters with limited-memory GPUs like 24 GB GPU GeForce RTX 4090, using METP can enable them to train or infer long-sequence data. From Fig. 4, we know that memory usage can easily exceed the limit. Applying METP can relieve us from buying expensive GPUs with larger memory.

### 6.4 Related works

Latest works such as Ring Attention (Liu H et al., 2023) and DeepSpeed-Ulysses (Jacobs et al., 2024) also aim to optimize the long-sequence computation of Transformers. Ring Attention applies send/recv to handle attention computation similar to our work. However, it achieves only up to 35% MFU, whereas ours achieves up to 48%. Different from their work, METP is a general parallelism method designed for $\boldsymbol{O} = f(\boldsymbol{AB})\boldsymbol{C}$, whereas we have specifically designed a two-level scheme for attention computation.

### 6.5 Limitations and future work

Our current implementation decomposes attention computation tasks into $p^3$ tasks, which leads to poor SM efficiency when the sequence length is small. Moreover, we do not consider overlap between layers yet. By implementing the METP CUDA kernel or using techniques like compiling (Chen et al., 2018), we may overcome these and achieve better performance in the future. Moreover, the MFU evaluation experiment reveals that METP is not universally

optimal. However, as shown in Tables 1 and 2, METP shares the same partitioning form with other methods. Given this, we are currently developing algorithms to dynamically select the most appropriate parallelism method based on varying sequence lengths.

## 7 Conclusions

In this paper, we introduce a parallelism scheme, namely METP, and present the detailed parallel algorithms for the generalized matrix multiplication computation $\boldsymbol{O} = f(\boldsymbol{AB})\boldsymbol{C}$. Our theoretical analysis demonstrates that METP can significantly reduce the memory consumption of attention to an $O(1/p^3)$ degree and reduce memory overheads of other intermediate results by at least 41.7%, which enables us to enlarge sequence length during training. Furthermore, we employ METP to address the memory problems encountered when training Transformer models with long sequences and provide implementations of METP MHA and METP FFN designed for Transformers. The experimental results demonstrate that our method outperforms existing strong baselines.

### Contributors

Peng LIANG designed this work. Peng LIANG, Linbo QIAO, Yanqi SHI, and Hao ZHENG conducted and analyzed the experiments. Yu TANG and Dongsheng LI reviewed and revised the paper. All the authors had participated in writing the paper.

### Acknowledgements

We would like to express our sincere gratitude to Zhixin OU and Zhiquan LAI from National University of Defense Technology for their assistance during the paper revision process.

### Conflict of interest

Dongsheng LI is a corresponding expert of *Frontiers of Information Technology & Electronic Engineering*, and he was not involved with the peer review process of this paper. All the authors declare that they have no conflict of interest.

### Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

### References

Achiam J, Adler S, Agarwal S, et al., 2023. GPT-4 technical report. https://doi.org/10.48550/arXiv.2303.08774

Beltagy I, Peters ME, Cohan A, 2020. Longformer: the long-document Transformer. https://doi.org/10.48550/arXiv.2004.05150

Brown TB, Mann B, Ryder N, et al., 2020. Language models are few-shot learners. Proc 34th Int Conf on Neural Information Processing Systems, Article 159.

Chen TQ, Moreau T, Jiang ZH, et al., 2018. TVM: an automated end-to-end optimizing compiler for deep learning. 13th USENIX Symp on Operating Systems Design and Implementation, p.578-594.

Chowdhery A, Narang S, Devlin J, et al., 2022. PaLM: scaling language modeling with pathways. *J Mach Learn Res*, 24(1):240.

Dao T, 2024. FlashAttention-2: faster attention with better parallelism and work partitioning. Proc 12th Int Conf on Learning Representations.

Dao T, Fu DY, Ermon S, et al., 2022. FlashAttention: fast and memory-efficient exact attention with IO-awareness. Proc 36th Int Conf on Neural Information Processing Systems, Article 1189.

Devlin J, Chang MW, Lee K, et al., 2019. BERT: pre-training of deep bidirectional Transformers for language understanding. Proc Conf of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, p.4171-4186. https://doi.org/10.18653/v1/N19-1423

Huang YP, Cheng YL, Bapna A, et al., 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. Proc 33rd Int Conf on Neural Information Processing Systems, Article 10.

Huang YP, Xu JW, Jiang ZX, et al., 2023. Advancing Transformer architecture in long-context large language models: a comprehensive survey. https://doi.org/10.48550/arXiv.2311.12351

Jacobs SA, Tanaka M, Zhang CM, et al., 2024. System optimizations for enabling training of extreme long sequence Transformer models. Proc 43rd ACM Symp on Principles of Distributed Computing, p.121-130. https://doi.org/10.1145/3662158.3662806

Kaddour J, Harris J, Mozes M, et al., 2023. Challenges and applications of large language models. https://doi.org/10.48550/arXiv.2307.10169

Kingma DP, Ba J, 2015. Adam: a method for stochastic optimization. Proc 3rd Int Conf on Learning Representations.

Korthikanti VA, Casper J, Lym S, et al., 2023. Reducing activation recomputation in large Transformer models. Proc 6th Conf on Machine Learning and Systems.

Lai ZQ, Li SW, Tang XD, et al., 2023. Merak: an efficient distributed DNN training framework with automated 3D parallelism for giant foundation models. *IEEE Trans Parall Distrib Syst*, 34(5):1466-1478. https://doi.org/10.1109/TPDS.2023.3247001

Li A, Song SL, Chen JY, et al., 2020. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Trans Parall Distrib Syst*, 31(1): 94-110. https://doi.org/10.1109/TPDS.2019.2928289

Li SG, Xue FZ, Baranwal C, et al., 2023. Sequence parallelism: long sequence training from system perspective. Proc 61st Annual Meeting of the Association for Computational Linguistics, p.2391-2404. https://doi.org/10.18653/v1/2023.acl-long.134

Liang P, Tang Y, Zhang XD, et al., 2023. A survey on auto-parallelism of large-scale deep learning training. *IEEE Trans Parall Distrib Syst*, 34(8):2377-2390. https://doi.org/10.1109/TPDS.2023.3281931

Liu H, Zaharia M, Abbeel P, 2023. Ring Attention with blockwise Transformers for near-infinite context. https://doi.org/10.48550/arXiv.2310.01889

Liu YX, Zhang K, Li Y, et al., 2024. Sora: a review on background, technology, limitations, and opportunities of large vision models. https://doi.org/10.48550/arXiv.2402.17177

Liu ZM, Cheng SG, Zhou HT, et al., 2023. Hanayo: harnessing wave-like pipeline parallelism for enhanced large model training efficiency. Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 56. https://doi.org/10.1145/3581784.3607073

Narayanan D, Shoeybi M, Casper J, et al., 2021a. Efficient large-scale language model training on GPU clusters using Megatron-LM. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 58. https://doi.org/10.1145/3458817.3476209

Narayanan D, Phanishayee A, Shi KY, et al., 2021b. Memory-efficient pipeline-parallel DNN training. Proc 38th Int Conf on Machine Learning, p.7937-7947.

Rajbhandari S, Rasley J, Ruwase O, et al., 2020. ZeRO: memory optimizations toward training trillion parameter models. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 20.

Rasley J, Rajbhandari S, Ruwase O, et al., 2020. DeepSpeed: system optimizations enable training deep learning models with over 100 billion parameters. Proc 26th ACM SIGKDD Int Conf on Knowledge Discovery & Data Mining, p.3505-3506. https://doi.org/10.1145/3394486.3406703

Shah J, Bikshandi G, Zhang Y, et al., 2024. FlashAttention-3: fast and accurate attention with asynchrony and low-precision. Proc 38th Int Conf on Neural Information Processing Systems.

Srivastava N, Hinton G, Krizhevsky A, et al., 2014. Dropout: a simple way to prevent neural networks from overfitting. *J Mach Learn Res*, 15(1):1929-1958.

Tarassow A, 2023. The potential of LLMs for coding with low-resource and domain-specific programming languages. https://doi.org/10.48550/arXiv.2307.13018

Touvron H, Lavril T, Izacard G, et al., 2023. LLaMA: open and efficient foundation language models. https://doi.org/10.48550/arXiv.2302.13971

Xu JJ, Sun X, Zhang ZY, et al., 2019. Understanding and improving layer normalization. Proc 33rd Int Conf on Neural Information Processing Systems, Article 394.

Zhao YL, Gu A, Varma R, et al., 2023. PyTorch FSDP: experiences on scaling fully sharded data parallel. *Proc VLDB Endow*, 16(12):3848-3860. https://doi.org/10.14778/3611540.3611569