BurstEngine: an Efficient Distributed Framework for Training Transformers on Extremely Long Sequences of over 1M Tokens

Ao Sun*
maydomine@bupt.edu.cn
Beijing University of Posts and Telecommunications, Beijing, China

Weilin Zhao*
zwl23@mails.tsinghua.edu.cn
Department of Computer Science and Technology, Tsinghua University, Beijing, China

Xu Han†
han-xu@tsinghua.edu.cn
Department of Computer Science and Technology, Tsinghua University, Beijing, China

Cheng Yang†
yangcheng@bupt.edu.cn
Beijing University of Posts and Telecommunications, Beijing, China

Zhiyuan Liu
Department of Computer Science and Technology, Tsinghua University, Beijing, China

Chuan Shi
Beijing University of Posts and Telecommunications, Beijing, China

Maosong Sun
Department of Computer Science and Technology, Tsinghua University, Beijing, China

ABSTRACT

Existing methods for training LLMs on long-sequence data, such as Tensor Parallelism and Context Parallelism, exhibit low Model FLOPs Utilization as sequence lengths and number of GPUs increase, especially when sequence lengths exceed 1M tokens. To address these challenges, we propose BurstEngine, an efficient framework designed to train LLMs on long-sequence data. BurstEngine introduces BurstAttention, an optimized distributed attention with lower communication cost than RingAttention. BurstAttention leverages topology-aware ring communication to fully utilize network bandwidth and incorporates fine-grained communication-computation overlap. Furthermore, BurstEngine introduces sequence-level selective checkpointing and fuses the language modeling head with the loss function to reduce memory cost. Additionally, BurstEngine introduces workload balance optimization for various types of attention masking. By integrating these optimizations, BurstEngine achieves a 1.2× speedup with much lower memory overhead than the state-of-the-art baselines when training LLMs on extremely long sequences of over 1M tokens. We have made our code publicly available on GitHub: https://github.com/thunlp/BurstEngine.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; **Neural networks**.

KEYWORDS

Transformer, Distributed Training, Large Language Model

*indicates equal contribution.
†indicates corresponding authors.

This work is licensed under a Creative Commons Attribution 4.0 International License.

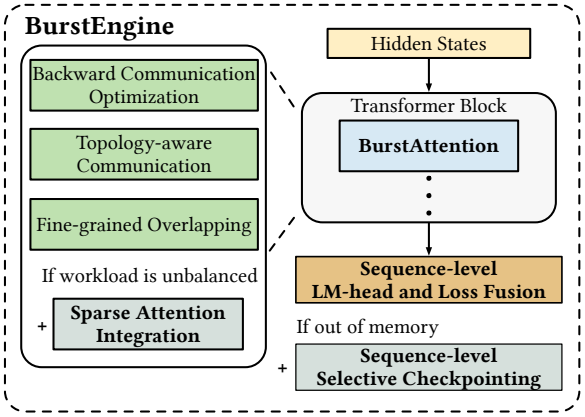

Figure 1: The overview of BurstEngine's main optimizations.

1 INTRODUCTION

Transformers [36] have emerged as the dominant architecture for large language models (LLMs) [3, 9, 34] and large multimodal models (LMMs) [5, 6, 43]. However, training Transformer-based models on long sequences faces two challenges: the memory consumption associated with storing large intermediate states and the quadratic cost of the attention mechanism with respect to sequence lengths. Considering that long-sequence training is crucial for LLMs and LMMs to generate extensive outputs such as documents, code, and videos, many efforts have been made to address the challenges associated with long-sequence training.

These efforts are approached from two perspectives: one focuses on improving the efficiency of individual devices (e.g., single GPU) in processing long sequences, while the other leverages distributed systems (e.g., multiple GPUs) to handle long sequences. On the one hand, typical works such as FlashAttention [7, 8] employ memory-friendly online softmax [26], enabling Transformer-based models

to efficiently process sequences of 32K tokens using a single device. Other works, such as gradient checkpointing[4, 27], employ recomputation to restore intermediate states rather than storing them, thus improving memory efficiency for long sequences. On the other hand, Context Parallelism (e.g., RingAttention) [13, 24], Tensor Parallelism [20, 27], and Head Parallelism (e.g., DeepSpeed-Ulysses) [18] are three typical approaches to leverage distributed clusters to train models on longer sequences. Recent LLMs and LMMs need to handle sequences beyond 128K tokens, making parallelism methods essential for training long-sequence Transformer-based models [1, 9, 17, 41]. In this paper, we propose "BurstEngine", an efficient framework specifically designed to train Transformer-based LLMs and LMMs on extremely long sequences of over 1M tokens. As illustrated in Figure 1, BurstEngine considers long-sequence optimizations from the perspective of the entire Transformer, especially handling the issues of attention for processing long sequences, including four parts: (1) **BurstAttention**, (2) **Sequence-level Selective Checkpointing**, (3) **Sequence-level Fusion of Language Modeling (LM) Head and Loss Function**, and (4) **Sparse Attention Integration**.

BurstAttention is a highly efficient distributed attention implementation. As illustrated in Figure 1, BurstAttention introduces three key optimizations: (1) Backward Communication Optimization, which reduces nearly 25% of communication cost in the backward pass compared to existing efficient context parallelism by rewriting the backward pass of attention modules in a communication-efficient way. (2) Topology-aware ring communication, which splits communication into intra-node and inter-node communication, and thus fully takes advantage of the bandwidth of different networks to reduce communication cost. (3) Fine-grained communication-computation overlap, which designs a specialized double buffer to better overlap computation and communication. BurstAttention significantly enhances Transformers' efficiency for long sequences, but long sequence bring high memory consumption due to storing intermediate states. To address this challenge, we propose sequence-level selective checkpointing and sequence-level fusion of LM head and loss function.

Sequence-level Selective Checkpointing is a novel gradient recomputation scheme specialized for attention modules, which optimizes the trade-off between memory consumption and computational overhead at the sequence level. Unlike traditional approaches [4] that store or recompute entire sequences, it selectively checkpoints the former part of a sequence, storing the latter part and recomputing the former during backward passes. This approach can significantly reduce memory consumption while maintaining limited computational cost.

Sequence-level Fusion of LM Head and Loss Function can reduce the memory consumption of the Language Model (LM) head by fusing the LM head with the cross-entropy loss function to reduce the memory consumption of storing massive intermediate states. Furthermore, we fuse the forward and backward passes of the LM head and cross-entropy loss function to avoid recomputing related intermediate states.

Sparse Attention Integration enables BurstEngine to flexibly and efficiently incorporate BurstAttention with a variety of sparse patterns, including causal masking, sliding-window masking, dilated masking, and other block-wise sparse patterns.

We evaluate BurstEngine on a series of settings with up to 64× GPUs. The experimental results show that BurstEngine achieves a 1.2× speedup compared to the state-of-the-art method on extremely long sequences of over 1M tokens. Additionally, BurstEngine saves 26.4% of memory compared to most memory-efficient baselines.

In summary, we make the following contributions: (1) We propose BurstAttention, a novel distributed attention with backward communication optimization, topology-aware ring communication pattern, and fine-grained overlapping. (2) We introduce a set of novel optimizations, including sequence-level selective checkpointing and sequence-level fusion of LM head and loss function, along with sparse attention integration. (3) We build BurstEngine, an implementation of the proposed framework with over 35K lines of Python, C++, and CUDA code, which achieves the state-of-the-art performance on training LLMs and LMMs on extremely long sequences of over 1M tokens.

2 PRELIMINARY

2.1 Preliminary of Transformers

A Transformer consists of multiple blocks, each with an attention module and a feed-forward network (FFN), along with a language modeling (LM) head mapping the final block's output to vocabulary space. In this section, we introduce the attention module and the LM head in detail as they are key bottlenecks in long-context training.

Transformer Attention Module. Given a sequence containing N tokens as input, whose embeddings are denoted as $\mathbf{X} \in \mathbb{R}^{N\times d}$, the attention module can be formalized as

$$\begin{aligned} \mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \mathbf{K} = \mathbf{X}\mathbf{W}_K, \mathbf{V} = \mathbf{X}\mathbf{W}_V, \\ \mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}, \mathbf{P} = \mathrm{Softmax}(\mathbf{S}), \mathbf{O} = \mathbf{P}\mathbf{V}, \mathbf{Y} = \mathbf{O}\mathbf{W}_{\mathrm{attn}}, \end{aligned} \tag{1}$$

where $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d\times d}$ map $\mathbf{X}$ to $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$, the query, key, and value embeddings. $\mathbf{S} \in \mathbb{R}^{N\times N}$ and $\mathbf{P} \in \mathbb{R}^{N\times N}$ indicate the attention scores and the attention probabilities, respectively. $\mathbf{O} \in \mathbb{R}^{N\times d}$ is the sum of value embeddings weighted by query-key similarities, mapped to $\mathbf{Y} \in \mathbb{R}^{N\times d}$ via $\mathbf{W}_{\mathrm{attn}} \in \mathbb{R}^{d\times d}$. Most LLMs and LMMs use multi-head attention, where each head follows Eq. (1) and outputs are concatenated. For simplicity, we omit multi-head details as the number of heads does not affect our optimizations and conclusions, and denote the process as $\mathbf{Y} = \mathrm{ATTN}(\mathbf{X})$. Since our efficiency methods apply broadly to various attention patterns, we refrain from detailed discussions on these patterns.

Transformer Block and LM head. A Transformer block $\mathbf{Y} = \mathrm{Transformer}(\mathbf{X})$ is given as

$$\mathbf{H} = \mathrm{ATTN}(\mathbf{X}) + \mathbf{X}, \mathbf{Y} = \mathrm{FFN}(\mathbf{H}) + \mathbf{H}. \tag{2}$$

After stacking M Transformer blocks, we can build an LLM or LMM to generate tokens

$$\mathrm{LM}(\mathrm{Transformer}_M(\cdots \mathrm{Transformer}_1(\mathbf{X}))), \tag{3}$$

where $\mathrm{LM}(\mathbf{X}) = \mathrm{Softmax}(\mathbf{X}\mathbf{W}_{\mathrm{vocab}}^\top)$ is the language modeling (LM) head that maps the outputs of a Transformer to the probability distribution, $\mathbf{W}_{\mathrm{vocab}} \in \mathbb{R}^{v\times d}$ is the vocabulary embeddings, and v is the size of the vocabulary set. For training LLMs and LMMs, the outputs of the LM head are used to calculate loss. For more details of Transformers, see [36] and surveys [14, 15, 19, 23].

2.2 Challenges of Processing Long Sequences in Transformers

As we mentioned before, the attention module and the LM head are key bottlenecks in training Transformers on long sequences. In this section, we explain why and how they become bottlenecks in training Transformers on long sequences.

(1) **Challenges in Attention Module.** The attention module exhibits quadratic complexity with respect to sequence lengths, making handling long sequences challenging in practice. As shown in Figure 2, attention modules have become the main bottleneck in training Transformer-based models on long sequences of over 128K tokens. Note that recent LLMs and LMMs are required to handle sequences of over 1M tokens. In this case, we have to use distributed clusters to make Transformers efficiently process long sequences.

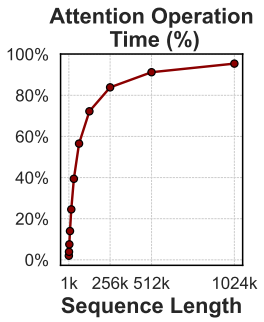

Figure 2: The proportion of time spent by attention modules during the end-to-end training process of a Transformer-based model (7B parameters).

(2) **Challenges of Storing Intermediate States.** Training Transformer models requires computing loss functions through forward passes and obtaining parameter gradients through backward passes. As sequence length increases, memory required to store intermediate states during the forward pass prior to performing the backward pass becomes a significant bottleneck. To address this, efficient memory optimization solutions are essential, especially for those GPUs with limited storage capacity but sufficient computing power.

(3) **Challenges in Language Modeling Head.** When computing the LM head, $\text{LM}(\mathbf{X}) = \text{Softmax}(\mathbf{X}\mathbf{W}_{\text{vocab}}^{\top})$, large memory is required to store the states $\mathbf{X}\mathbf{W}_{\text{vocab}}^{\top} \in \mathbb{R}^{N\times v}$ before the Softmax function. This memory consumption continues to grow as the sequence length increases, eventually reaching a point where it exceeds the capacity of a single GPU.

(4) **Challenges of Integrating Sparse Attention.** In LLMs and LMMs, attention modules are often coupled with sparse masking to control which keys and values participate in the attention mechanism. In other words, the probabilities in Eq. (1) are obtained by $\mathbf{P} = \text{Softmax}(\mathbf{M} \odot \mathbf{S})$, where $\mathbf{M} \in \mathbb{R}^{N\times N}$ is the masking matrix. For example, in LLMs, each token is required to only attend to the tokens preceding it in the sequence, which necessitates the use of a triangular masking matrix. Additionally, sparse attention mechanisms [37, 40] are often employed to speed up long-sequence processing. These sparse mechanisms rely on a sparse masking matrix during the attention process to reduce computational complexity. Obviously, the introduction of complex masking leads to an imbalanced workload, presenting challenges for using distributed clusters to process long sequences.

In subsequent sections, we will focus on introducing **how to address these challenges on distributed clusters and efficiently train Transformer-based models on long sequences**.

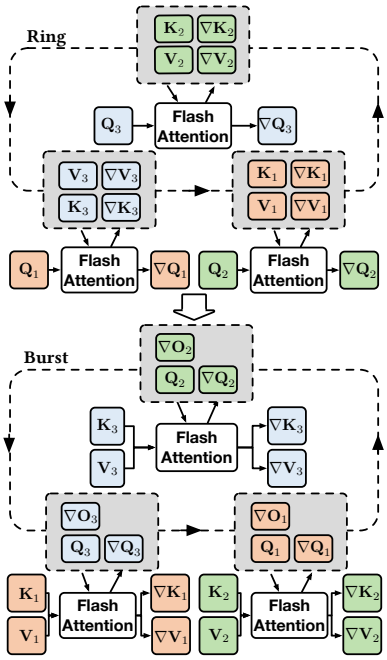

Figure 3: The backward pass process of RingAttention and BurstAttention using 3 GPUs.

3 OVERALL FRAMEWORK OF BURSTENGINE

3.1 BurstAttention

BurstAttention addresses the efficiency challenges encountered by the attention module during the training of Transformers on long-sequence data by leveraging a distributed cluster. To facilitate introducing the details of BurstAttention, here we define a distributed cluster as a cluster built by several nodes, and each node contains several GPUs. Similar to the recent **competitive Context Parallelism method RingAttention**, after obtaining $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in Eq. (1), BurstAttention divides these sequence embeddings into multiple partitions along the sequence dimension according to the number of GPUs in the cluster. Each GPU is assigned a query partition, a key partition, and a value partition. Formally, given the GPU number G of the cluster, the i-th GPU is assigned $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i \in \mathbb{R}^{\frac{N}{G}\times d}$. Since the parallelism efficiency primarily depends on minimizing communication costs while maximizing the overlap between communication and computation, BurstAttention introduces three optimizations to reduce communication cost and achieve better scalability and efficiency: backward communication optimization, topology-aware ring communication, and fine-grained overlapping of communication and computation. Next, we will first introduce how BurstAttention completes basic forward passes, and then introduce three optimizations in detail.

Forward Pass of BurstAttention. BurstAttention formalizes the forward pass of an attention module into a multi-step process. At each step, the i-th GPU receives $\mathbf{K}_j$ and $\mathbf{V}_j$ from its neighbor and performs local attention operations, and these local operations

Algorithm 1 The backward pass of RingAttention

Require: Matrices $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i, \mathbf{O}_i, \nabla\mathbf{O}_i \in \mathbb{R}^{\frac{N}{G}\times d}$, $\mathbf{Lse}_i \in \mathbb{R}^{\frac{N}{G}}$ on the i-th GPU

Initialize $\nabla\mathbf{Q}_i, \nabla\mathbf{K}_i, \nabla\mathbf{V}_i = (0)_{\frac{N}{G}\times d}$
Put $\mathbf{K}_i, \mathbf{V}_i, \nabla\mathbf{K}_i, \nabla\mathbf{V}_i$ into communication ring
for $j = 1$ to G **do**
  Conduct one step of the ring communication;
  Get $\mathbf{K}_j, \mathbf{V}_j, \nabla\mathbf{K}_j, \nabla\mathbf{V}_j$ from communication ring;
  $\mathbf{S}_{i,j} = \mathbf{Q}_i\mathbf{K}_j^\top$;
  $\mathbf{P}_{i,j} = \exp(\mathbf{S}_{i,j} - \mathbf{Lse}_i)$;
  $\nabla\mathbf{V}_j = \nabla\mathbf{V}_j + \mathbf{P}_{i,j}^\top\nabla\mathbf{O}_i$;
  $\nabla\mathbf{P}_{i,j} = \nabla\mathbf{O}_i\,\mathbf{V}_j^\top$;
  $\mathbf{D}_i = \text{rowsum}(\nabla\mathbf{O}_i \circ \mathbf{O}_i)$
  $\nabla\mathbf{S}_{i,j} = \mathbf{P}_{i,j} \circ (\nabla\mathbf{P}_{i,j} - \mathbf{D}_i)$;
  $\nabla\mathbf{K}_j = \nabla\mathbf{K}_j + \nabla\mathbf{S}_{i,j}^\top\mathbf{Q}_i$;
  $\nabla\mathbf{Q}_i = \nabla\mathbf{Q}_i + \nabla\mathbf{S}_{i,j}\,\mathbf{K}_j$;
  Put $\mathbf{K}_j, \mathbf{V}_j, \nabla\mathbf{K}_j, \nabla\mathbf{V}_j$ into communication ring; {4Nd}
output $\nabla\mathbf{Q}_i, \nabla\mathbf{K}_i, \nabla\mathbf{V}_i$;

Algorithm 2 The backward pass of BurstAttention

Require: Matrices $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i, \mathbf{O}_i, \nabla\mathbf{O}_i \in \mathbb{R}^{\frac{N}{G}\times d}$, $\mathbf{Lse}_i \in \mathbb{R}^{\frac{N}{G}}$ on the i-th GPU

Initialize $\nabla\mathbf{Q}_i, \nabla\mathbf{K}_i, \nabla\mathbf{V}_i = (0)_{\frac{N}{G}\times d}$
$D_i = \text{rowsum}(\nabla\mathbf{O}_i \circ \mathbf{O}_i)$
Put $\mathbf{Q}_i, \nabla\mathbf{Q}_i, \nabla\mathbf{O}_i, \mathbf{D}_i, \mathbf{Lse}_i$ into communication ring
for $j = 1$ to G **do**
  Conduct one step of the ring communication;
  Get $\mathbf{Q}_j, \nabla\mathbf{Q}_j, \nabla\mathbf{O}_j, \mathbf{D}_j, \mathbf{Lse}_j$ from communication ring;
  $\mathbf{S}_{j,i} = \mathbf{Q}_j\mathbf{K}_i^\top$;
  $\mathbf{P}_{j,i} = \exp(\mathbf{S}_{j,i} - \mathbf{Lse}_j)$;
  $\nabla\mathbf{V}_i = \nabla\mathbf{V}_i + \mathbf{P}_{j,i}^\top\nabla\mathbf{O}_j$;
  $\nabla\mathbf{P}_{j,i} = \nabla\mathbf{O}_j\,\mathbf{V}_i^\top$;
  $\nabla\mathbf{S}_{j,i} = \mathbf{P}_{j,i} \circ (\nabla\mathbf{P}_{j,i} - \mathbf{D}_j)$;
  $\nabla\mathbf{K}_i = \nabla\mathbf{K}_i + \nabla\mathbf{S}_{j,i}^\top\mathbf{Q}_j$;
  $\nabla\mathbf{Q}_j = \nabla\mathbf{Q}_j + \nabla\mathbf{S}_{j,i}\,\mathbf{K}_i$;
  Put $\mathbf{Q}_j, \nabla\mathbf{Q}_j, \nabla\mathbf{O}_j, \mathbf{D}_j, \mathbf{Lse}_j$ into communication ring; {3Nd + 2N}
output $\nabla\mathbf{Q}_i, \nabla\mathbf{K}_i, \nabla\mathbf{V}_i$;

can be formalized as

$$\mathbf{S}_{i,j} = \frac{\mathbf{Q}_i\mathbf{K}_j^\top}{\sqrt{d}}, \mathbf{P}_{i,j} = \text{Softmax}(\mathbf{S}_{i,j}), \mathbf{O}_{i,j} = \mathbf{P}_{i,j}\mathbf{V}_j, \tag{4}$$

where $\mathbf{O}_{i,j} \in \mathbb{R}^{\frac{N}{G}\times d}$ indicates the local hidden states. $\mathbf{S}_{i,j} \in \mathbb{R}^{\frac{N}{G}\times\frac{N}{G}}$ and $\mathbf{P}_{i,j} \in \mathbb{R}^{\frac{N}{G}\times\frac{N}{G}}$ indicate the local attention scores and the local attention probabilities, respectively. After local attention operations, the i-th GPU sends $\mathbf{K}_j$ and $\mathbf{V}_j$ to its next neighbor for the next step. Within each single GPU, we adopt FlashAttention to efficiently complete local attention operations. For more details about FlashAttention, please refer to its related papers [7, 8].

This multi-step process continues until all $\mathbf{K}$ and $\mathbf{V}$ partitions have gone around the ring and completed all local attention operations. Generally, after obtaining all local states $\{\mathbf{O}_{i,j}\}_{i=1,j=1}^{G,G}$, we have to introduce additional communication and computation to aggregate these states into global states, as well as a lot of memory consumption to store these local states before aggregating them. To avoid incurring additional overhead, during the ring process, we introduce online softmax [26] to progressively aggregate all local states $\{\mathbf{O}_{i,j}\}_{j=1}^{G}$ into the partitioned global states $\mathbf{O}_i \in \mathbb{R}^{\frac{N}{G}\times d}$, and finally map the concatenated partitioned global states $\{\mathbf{O}_i\}_{i=1}^{G}$ to the output embeddings $\mathbf{Y} \in \mathbb{R}^{N\times d}$. Specifically, we reformulate the local operations in Eq. (4) as

$$\begin{aligned}\mathbf{S}_{i,j} &= \frac{\mathbf{Q}_i\mathbf{K}_j^\top}{\sqrt{d}}, \mathbf{Lse} = \text{LSE}(\mathbf{S}_{i,j}),\\ \mathbf{P}_{i,j} &= \exp\left(\mathbf{S}_{i,j} - \mathbf{Lse}\right), \mathbf{O}_{i,j} = \mathbf{P}_{i,j}\mathbf{V}_j.\end{aligned} \tag{5}$$

where LogSumExp (LSE) function maps $\mathbb{R}^{\frac{N}{G}\times\frac{N}{G}}$ to $\mathbb{R}^{\frac{N}{G}}$ and given

$$[\mathbf{Y}]_i = \log\sum_{j=1}^{\frac{N}{G}}\exp([\mathbf{X}]_{i:j}), \tag{6}$$

where $[\mathbf{Y}]_i$ is the i-th element of the vector $\mathbf{Y}$, $[\mathbf{X}]_{i:j}$ is the element located in the i-th row and the j-th column of the matrix $\mathbf{X}$. With Eq. (5), we can avoid storing $\{\mathbf{S}_{i,j}\}_{i=1,j=1}^{G,G}$, $\{\mathbf{P}_{i,j}\}_{i=1,j=1}^{G,G}$, $\{\mathbf{O}_{i,j}\}_{i=1,j=1}^{G,G}$ to obtain $\mathbf{O}_i$. Note that, the subtraction of the matrix $\mathbf{S}_{i,j}$ and the vector $\mathbf{Lse}$ requires broadcasting the elements of $\mathbf{Lse}$ along the last dimension of $\mathbf{S}_{i,j}$, and we will not repeatedly emphasize this.

Communication Optimization of Backward Pass. Given a sequence consisting of N tokens, whether using RingAttention or BurstAttention, the i-th GPU has $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i \in \mathbb{R}^{\frac{N}{G}\times d}$ and the partitioned global states $\mathbf{O}_i \in \mathbb{R}^{\frac{N}{G}\times d}$ after finishing forward pass. The communication cost of forward pass is $2Nd$, because each GPU receives and sends $\{\mathbf{K}_j\}_{j=1}^{G}$ and $\{\mathbf{V}_j\}_{j=1}^{G}$ once.

As shown in Figure 3 and Algorithm 1, during the backward pass of RingAttention, $\mathbf{K}_j, \mathbf{V}_j$ and their corresponding gradients $\nabla\mathbf{K}_j, \nabla\mathbf{V}_j$ are passed around the ring. At each step, the i-th GPU receives $\mathbf{K}_j, \mathbf{V}_j, \nabla\mathbf{K}_j, \nabla\mathbf{V}_j$ from its previous neighbor, and then use $\mathbf{K}_j, \mathbf{V}_j$ and the locally stored $\mathbf{Q}_i, \mathbf{O}_i, \nabla\mathbf{O}_i$ to update $\nabla\mathbf{Q}_i, \nabla\mathbf{K}_j, \nabla\mathbf{V}_j$. After updating gradients, the i-th GPU sends the received partitions $\mathbf{K}_j, \mathbf{V}_j$ and the updated gradients $\nabla\mathbf{K}_j, \nabla\mathbf{V}_j$ to its next neighbor for the next step. It is evident that for RingAttention, the communication cost during the backward pass amounts to $4Nd$ per GPU, doubling the cost incurred during the forward pass.

To reduce the communication cost of the backward pass, BurstAttention adopts a different strategy from RingAttention. Specifically, since $\mathbf{P}_i = \text{Softmax}(\mathbf{S}_i)$ and $\nabla\mathbf{P}_i = \nabla\mathbf{O}_i\mathbf{V}^\top$, we can get

$$\begin{aligned}\nabla\mathbf{S}_i &= \mathbf{P}_i \circ \nabla\mathbf{P}_i - \mathbf{P}_i\nabla\mathbf{P}_i^\top\mathbf{P}_i = \mathbf{P}_i \circ \nabla\mathbf{P}_i - \mathbf{P}_i(\nabla\mathbf{O}_i\mathbf{V}^\top)^\top\mathbf{P}_i\\ &= \mathbf{P}_i \circ \nabla\mathbf{P}_i - (\mathbf{P}_i\mathbf{V})\nabla\mathbf{O}_i^\top\mathbf{P}_i = \mathbf{P}_i \circ \nabla\mathbf{P}_i - \mathbf{O}_i\nabla\mathbf{O}_i^\top\mathbf{P}_i,\end{aligned} \tag{7}$$

where $\circ$ denotes element-wise multiplication. Given $\mathbf{D}_i = \mathbf{O}_i\nabla\mathbf{O}_i^\top$,

$$\nabla\mathbf{S}_i = \mathbf{P}_i \circ \nabla\mathbf{P}_i - \mathbf{D}_i\mathbf{P}_i. \tag{8}$$

Based on the above derivation, BurstAttention stores $\mathbf{K}_i, \mathbf{V}_i, \nabla\mathbf{K}_i, \nabla\mathbf{V}_i$ on each GPU, and passes $\mathbf{Q}_j, \nabla\mathbf{Q}_j, \nabla\mathbf{O}_j, \mathbf{Lse}_j, \mathbf{D}_j$ around the ring. Through formula substitution, we can pass $\mathbf{D}_j$ instead of $\mathbf{O}_j$ around the ring, and the whole backward pass of BurstAttention is shown in Algorithm 2. Since the total size of $\{\mathbf{D}_j\}_{j=1}^{G}$ and $\{\mathbf{Lse}_j\}_{j=1}^{G}$ is N, we can see that the communication cost of backward pass is $3Nd + 2N$ for BurstAttention, nearly 25% less than RingAttention's $4Nd$. In addition, the computation overhead of BurstAttention is also lower than that of other RingAttention since we do not need to calculate $\text{rowsum}(\nabla\mathbf{O}_i \circ \mathbf{O}_i)$ in each round.

Topology-aware Ring Communication and Fine-grained Communication-Computation Overlapping. Traditional Context Parallelism [13, 22, 24] experiences serious performance degradation in multi-node settings because of the limited communication bandwidth of inter-node networks. To address this, DoubleRing

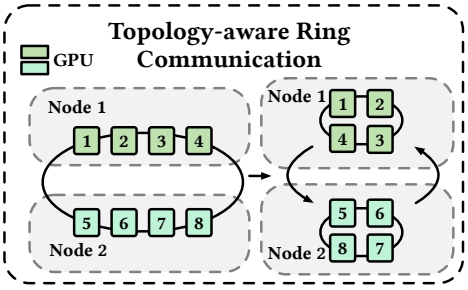

Figure 4: An example of topology-aware ring communication using 2×4 GPUs.

Attention [13] partitions global ring communication into intra-node and inter-node communication. This approach enhances the utilization of inter-node network interface controllers (NICs) and benefits from the overlapping of heterogeneous communication. While DoubleRing Attention offers notable advantages by overlapping inter-node communication, intra-node communication, and computation during forward passes, it fails to overlap gradient communication during backward passes. This oversight inevitably leads to performance degradation as the cost of gradient communication grows. To overcome these limitations, we introduce the topology-aware ring communication with fine-grained overlapping.

Table 1: The communication time (forward and backward pass) of RingAttention, DoubleRingAttention, and BurstAttention. $\mathbf{T_{intra}} = \mathbf{Lat_{intra}} + \frac{P}{B_{\mathbf{intra}}}$. $\mathbf{T_{inter}} = \mathbf{Lat_{inter}} + \frac{P}{B_{\mathbf{inter}}}$. $\mathbf{Lat_{intra}}$ and $\mathbf{Lat_{inter}}$ are the latencies of intra-node communication and inter-node communication, respectively. $B_{\mathbf{intra}}$ and $B_{\mathbf{inter}}$ are the bandwidths of intra-node communication and inter-node communication, respectively. P is the size of data to be communicated, $N_{\mathbf{intra}}$ is the number of GPUs in the same node, $N_{\mathbf{inter}}$ is the node number, and G is total number of GPUs in all nodes.

Method	Communication Time
RingAttention	$6\max(N \cdot \mathrm{T}_{\mathrm{intra}}, N \cdot \mathrm{T}_{\mathrm{inter}})$
DoubleRing	$4\max((N - N_{\mathrm{inter}}) \cdot \mathrm{T}_{\mathrm{intra}}, N_{\mathrm{inter}} \cdot \mathrm{T}_{\mathrm{inter}})$ $+2((N - N_{\mathrm{inter}}) \cdot \mathrm{T}_{\mathrm{intra}} + N_{\mathrm{inter}} \cdot \mathrm{T}_{\mathrm{inter}})$
BurstAttention	$5\max((N - N_{\mathrm{inter}}) \cdot \mathrm{T}_{\mathrm{intra}}, N_{\mathrm{inter}} \cdot \mathrm{T}_{\mathrm{inter}})$

As Figure 4 shows, in a 2-node cluster, where each node contains 4 GPUs interconnected via NVLink, and the nodes are interconnected with the InfiniBand network, BurstAttention divides the global ring into two sub-rings, each responsible for communication among the GPUs within the same node. During each round of communication, GPUs within the same node perform a ring communication operation to exchange data. After all GPUs within the same node have completed their data exchanges (4 rounds intra-node communication in this case), GPUs in different nodes exchange data via the global ring. Considering there is commonly more than one NIC in distributed clusters for LLM training, the global ring communication can effectively utilize all available NICs' bandwidth and reduce inter-node communication cost significantly.

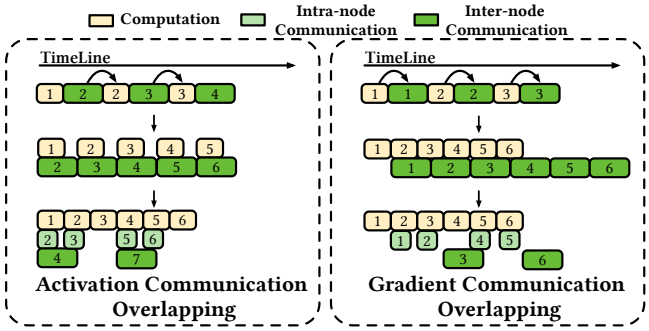

Figure 5: The communication overlap for activations and gradients in BurstAttention.

Since intra-node and inter-node communication in Figure 4 have separate railways in NVLink and InfiniBand networks, respectively, we can further overlap inter-node communication and intra-node communication. To achieve this, BurstAttention uses three dedicated buffers based on the topology: one for intra-node communication, one for inter-node communication, and one for computation. In each communication round, BurstAttention swaps the intra-node communication buffer with the computation buffer, allowing GPUs to obtain the data transmitted in the previous round for attention computation while continuing data exchange in this round. After all intra-node communication is complete, the inter-node communication buffer is swapped with the computation buffer, enabling GPUs to access data from remote nodes while initiating the next round of inter-node communication.

To achieve fine-grained overlapping between computation, intra-node communication, and inter-node communication, it is essential to properly manage the dependencies between communication and computation. Specifically, as illustrated in Figure 5, we categorize overlapping of BurstAttention into two types: 1) Activation (e.g., $\mathbf{K}, \mathbf{V}$) overlapping, where the first round of computation can be scheduled to execute concurrently with communication, as each round of communication is independent of the computation in the corresponding round; 2) Gradient (e.g., $\nabla\mathbf{Q}$) overlapping, where the first round of communication has a dependency for the computation of the corresponding round, thus we have to first compute the gradients and then communicate these gradients.

As illustrated in Figure 5, for activation overlapping, BurstAttention overlaps computation and communication by launching inter-node and intra-node communication threads simultaneously with computation threads. After one round of computation, BurstAttention waits for intra-node communication and launches another computation thread using received activations. After each device fully exchanges activations with other devices within the same node, it waits until inter-node communication is finished and then launches computation thread using received activations. The overlapping method for activations is not applicable for gradients since the first round of communication has a dependency on the gradient computation. To solve this issue, as shown in Figure 5, BurstAttention delays intra-node communication and inter-node communication. Specifically, BurstAttention first performs one round of

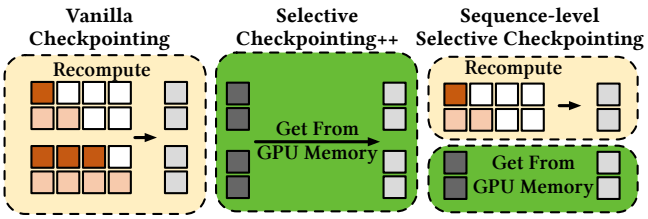

Figure 6: The illustration of sequence-level selective checkpointing in BurstEngine.

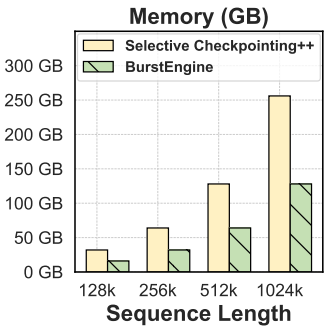

Figure 7: The total memory consumption of different gradient checkpointing strategies.

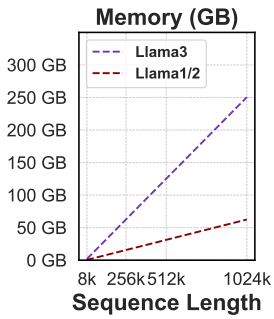

Figure 8: The total memory cost of LM head's logits for LLaMA-1/2 and LLaMA-3.

computation for warm-up. Then it initiates the intra-node communication and another round of computation, which gets rid of the dependency between later gradient communication and gradient computation. After all GPUs have exchanged the gradients with each other, BurstAttention launches the inter-node communication to exchange the gradients between different nodes. In this way, BurstAttention can almost fully overlap the computation, intra-node communication, and inter-node communication of both forward and backward passes.

As shown in Table 1, BurstAttention has less communication time than RingAttention when B_{intra} is larger than B_{inter}. By comparing the communication time of DoubleRingAttention and BurstAttention, we can find that BurstAttention not only has less communication cost due to backward communication optimization but also can save more communication time by overlapping inter-node communication and intra-node communication for gradients.

3.2 Sequence-level Selective Checkpointing

As the sequence length increases, the memory required to store intermediate states becomes a significant bottleneck. To address this, gradient checkpointing [4] is introduced to mitigate the high memory demands associated with storing these intermediate states. Generally, gradient checkpointing methods only store the input of each Transformer layer and get all intermediate states of each Transformer layer by using the input to do recomputation.

While gradient checkpointing methods reduce memory consumption, they increase computation overhead when combined with FlashAttention [7]. This is mainly because FlashAttention [7] fuses the Softmax process and matrix multiplications of the attention module, which means all intermediate states in attention would not be saved in forward passes and need to be recomputed in backward passes. To mitigate this issue, selective checkpointing++ [13, 21] stores FlashAttention's output in memory and includes the outputs in a whitelist, avoiding recomputation and directly using the outputs stored in GPU memory. However, this method would introduce substantial memory consumption since the outputs of FlashAttention are large as the sequence length increases and the number of layers increases, as shown in Figure 7.

To achieve a balance between memory consumption and computation overhead, we propose sequence-level selective checkpointing. As illustrated in Figure 6, the cost of recomputation for different sequence segments in the attention module is typically uneven, whereas the cost of storing activations remains the same in LLMs. Based on this observation, sequence-level selective checkpointing employs gradient checkpointing by dividing the sequence into two segments, storing the second segment with higher recomputation overhead, and only recomputing the first segment, as illustrated in Figure 6. In this manner, our method exhibits lower memory consumption with minimal additional computation overhead than other methods. As shown in Figure 7, sequence-level selective checkpointing can reduce the memory consumption of gradient checkpointing by 50% compared to selective gradient checkpointing++ while only introducing a slight decrease in end-to-end throughput.

3.3 Sequence-level Fusion of Language Modeling Head and Loss Function

In this section, we explain why the LM head becomes a key bottleneck in Transformers' long sequence training and introduce the details of the sequence-level fusion of the LM head and loss function. First, we dive into the details of the LM head. Here, we refine Eq. (3) into

$$\begin{aligned} \textbf{Logits} &= \mathbf{H}_{\text{last}}\mathbf{W}_{\text{head}}^{\top}, \mathbf{P} = \text{Softmax}(\textbf{Logits}), \\ \mathcal{L} &= \text{Cross-Entropy}(\mathbf{P}, \mathbf{Y}), \end{aligned} \tag{9}$$

where $\mathbf{H}_{\text{last}} \in \mathbb{R}^{N\times d}$ is the output embeddings of the last Transformer layer, $\mathbf{W}_{\text{head}} \in \mathbb{R}^{v\times d}$ is the vocabulary weight of the LM head, $\mathbf{P}$ is the probability distribution over the token vocabulary, and $\mathbf{Y}$ is the ground truth of the input.

To better support multiple languages and tasks, several LLMs have expanded their vocabulary sizes [9, 17]. Taking LLaMA [9, 33, 34] as an example, the vocabulary size of the LLaMA series has grown from 32K in LLaMA-1 and LLaMA-2 [33, 34] to 128K in LLaMA-3 [9]. As the sequence length increases, the memory consumption of storing the outputs of the LM head becomes a significant bottleneck, especially with a large vocabulary size. As shown in Figure 8, the memory consumption increases linearly with the sequence length, and the memory consumption of the LM head of LLaMA-3 is astonishingly high when dealing with long sequences. To address this issue, works like [25, 39] are proposed to split the hidden states and weights of the LM head into tiles, then fuse the LM head and cross-entropy at the tile level. In this way, there is no need to store the whole $\mathbf{P}$ matrix. During the backward pass, the above methods recompute the $\mathbf{P}$ matrix in the same way

Algorithm 3 Sequence-level fusion of LM head and loss function

Require: $\mathbf{H}_{\text{last}} \in \mathbb{R}^{N\times d}, \mathbf{W}_{\text{head}} \in \mathbb{R}^{v\times d}, \mathbf{Y} \in \mathbb{R}^{N}$
Block Size B_s, B_v
$\mathbf{W}_{\text{target}} = \mathbf{W}_{\text{head}}[\mathbf{Y}]$ {Index Operation, $\mathbf{W}_{\text{target}} \in \mathbb{R}^{N\times d}$}
{Divide $\mathbf{H}_{\text{last}}, \mathbf{W}_{\text{head}}, \mathbf{Y}$ into the tiles of the size $B_s \times d$, $B_v \times$ d and B_s}
for blocks $\mathbf{H}_i \in \mathbb{R}^{B_s\times d}$ in $\mathbf{H}_{\text{last}}$ **do**
  $\mathbf{Lse}_i = -\infty$;
  for blocks $\mathbf{W}_j \in \mathbb{R}^{B_v\times d}$ in $\mathbf{W}_{\text{head}}$ **do**
    $\mathbf{Logits}_{ij} = \mathbf{H}_i\mathbf{W}_j^\top$;
    $\mathbf{Lse}_{ij} = \log\sum\exp(\mathbf{Logits}_{ij})$;
    $\mathbf{Lse}_i = \log(\exp(\mathbf{Lse}_i) + \exp(\mathbf{Lse}_{ij}))$;
  $\mathcal{L}_i = -\sum \mathbf{H}_i \cdot \mathbf{W}_{\text{target}} + \mathbf{Lse}_i$;
  for blocks $\mathbf{W}_j \in \mathbb{R}^{B_v\times d}$ in $\mathbf{W}_{\text{head}}$ **do**
    $\nabla\mathbf{Logits}_{ij} = \exp(\mathbf{Logits}_{ij} - \mathbf{Lse}_i)$;
    $\mathbf{Index}_j = [jB_v + 1, \cdots, (j+1)B_v]$;
    $\mathbf{E} = 1$ if $\mathbf{Index}_j = \mathbf{Y}_i$, otherwise 0; {$\mathbf{E} \in \mathbb{R}^N$}
    $\nabla\mathbf{H}_i$ += $(\nabla\mathbf{Logits}_{ij} + \mathbf{E})\mathbf{W}_j^\top$.
    $\nabla\mathbf{W}_j$ += $(\nabla\mathbf{Logits}_{ij} + \mathbf{E})^\top\mathbf{H}_i$.

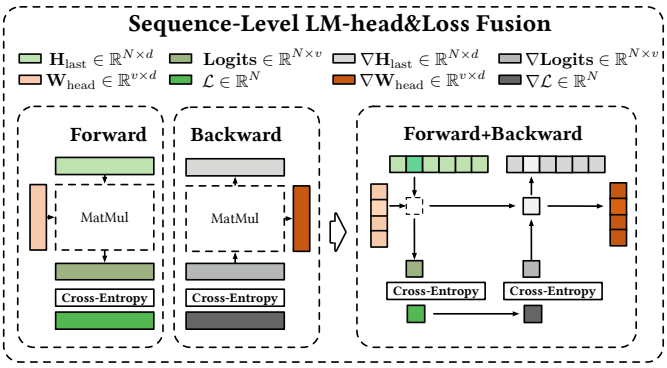

Figure 9: The illustration of the sequence-level fusion of the LM head and loss function.

as the forward pass. However, these methods still introduce unnecessary computation overhead since they need to recompute the **P** in backward passes and would suffer from low training throughput.

To achieve better efficiency in training Transformers on long-sequence data, we introduce a sequence-level fusion of LM head and loss function without the need to recompute **P** and **Logits**. As illustrated in Figure 9, we propose to fuse the forward pass and backward pass in LM head and loss fusion. In detail, we divide $\mathbf{H}_{\text{last}}$ into tiles along the sequence dimension and $\mathbf{W}_{\text{head}}$ into tiles along the vocabulary dimension. During the forward pass, we loop over each tile of $\mathbf{H}_{\text{last}}$ and $\mathbf{W}_{\text{head}}$, and compute **Logits** and update **Lse** for each tile. After that, we compute the loss function for each tile of $\mathbf{H}_{\text{last}}$ using **Lse** of **Logits**. Then, we perform the backward pass immediately after the forward pass, without the need to recompute **logits** and **P**, which is the same idea as the online-softmax part of BurstAttention, as mentioned before. During the backward pass, we compute the $\nabla\mathbf{Logits}$, $\nabla\mathbf{H}_{\text{last}}$, $\nabla\mathbf{W}_{\text{head}}$ using tiles of $\mathbf{H}_{\text{last}}$ and $\mathbf{W}_{\text{head}}$. In this way, the sequence-level fusion of the LM head and loss function can reduce most memory consumption of storing the outputs of the LM head and avoid unnecessary computation overhead in the backward pass, as depicted in Algorithm 3.

3.4 Sparse Attention Integration

In this section, we discuss how BurstAttention integrates with sparse attention patterns, including common causal attention masking and other block-wise sparse attention masking.

Causal Attention. In the training process of most LLMs and LMMs, the goal is to predict the next token in the sequence. In this task, the training objective is to maximize the log-likelihood of the next token given the previous tokens. To accomplish this, Transformer-based models commonly employ causal attention masking. For each token in the sequence, causal attention masking makes the token only attend to the key-value pairs of its preceding tokens. We can formulate this causal attention pattern as

$$\mathbf{O}_i = \text{Softmax}\left(\frac{\mathbf{Q}_i[\mathbf{K}_1, \mathbf{K}_2, \cdots, \mathbf{K}_i]^\top}{\sqrt{d}}\right)[\mathbf{V}_1, \mathbf{V}_2, \cdots, \mathbf{V}_i] \tag{10}$$

where $\mathbf{Q}_i$ is the query embedding of the i-th token, and $\mathbf{K}_j, \mathbf{V}_j$ are the key and value embeddings of the j-th token.

To balance the workload of causal attention in Context Parallelism [13, 24], two typical methods [2, 28, 44] have been proposed: striped-way workload balance and zigzag-way workload balance, as illustrated in Figure 10. Next, we will introduce how BurstAttention integrates these two kinds of methods to balance workloads.

For zigzag-way workload balance, the input sequence is initially partitioned along the sequence dimension across multiple devices. Given a sequence containing N tokens $\{x_1, \ldots, x_n\}$ and G devices, the sequence is divided into $2G$ subsequences, where the subsequence length is $P = \frac{N}{2G}$. For the i-th device ($1 \le i \le G$), it obtains two subsequences, one in the front and one in the back: S_i^1 and S_i^2,

$$\begin{aligned} S_i^1 &= \{x_k \mid k \in [(i-1)\times P + 1, i\times P]\}, \\ S_i^2 &= \{x_k \mid k \in [n - i\times P + 1, n - (i-1)\times P]\}. \end{aligned} \tag{11}$$

After getting subsequences, each device first performs causal attention on the two subsequences and then communicates with other devices to obtain the key-value. Based on the partition, the i-th device has a front query and a back query, after getting the front key-value and back key-value from other devices. When the i-th device has the query embeddings $\{\mathbf{Q}_i^1, \mathbf{Q}_i^2\}$ of S_i^1, S_i^2 and receives $\{\mathbf{K}_j^1, \mathbf{K}_j^2\}, \{\mathbf{V}_j^1, \mathbf{V}_j^2\}$ of the j-th device's S_j^1, S_j^2, we get

$$\mathbf{O}_{ij} = \begin{cases} \text{CausalATTN}(\{\mathbf{Q}_i^1, \mathbf{Q}_i^2\}, \{\mathbf{K}_j^1, \mathbf{K}_j^2\}, \{\mathbf{V}_j^1, \mathbf{V}_j^2\}) & \text{if } i = j, \\ \text{FullATTN}(\mathbf{Q}_i^2, \{\mathbf{K}_j^1, \mathbf{K}_j^2\}, \{\mathbf{V}_j^1, \mathbf{V}_j^2\}) & \text{if } i < j, \\ \text{FullATTN}(\{\mathbf{Q}_i^1, \mathbf{Q}_i^2\}, \mathbf{K}_j^1, \mathbf{V}_j^1) & \text{if } i > j. \end{cases} \tag{12}$$

For striped-way workload balance, the input sequence is initially partitioned along the sequence dimension across multiple devices into G subsequences, where the subsequence length is $P = \frac{N}{G}$ For the i-th device ($1 \le i \le G$), it obtains one subsequence as

$$S_i = \{x_k \mid k \in \{i + G\times m \mid m \in [0, P-1]\}\}. \tag{13}$$

The striped-way workload balance method ensures that each device can perform causal attention on the same number of tokens. When the i-th device has the query embeddings $\mathbf{Q}_i$ of S_i and receives the

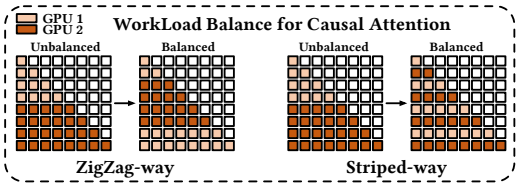

Figure 10: Two types of workload balance methods for distributed causal attention.

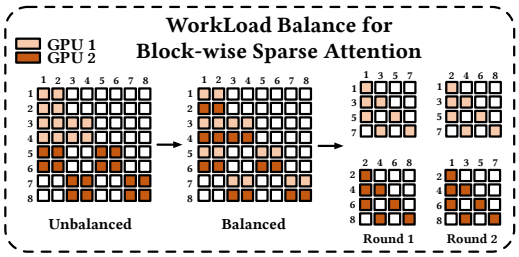

Figure 11: Workload balance for block-wise sparse attention.

key-value embeddings $\mathbf{K}_j, \mathbf{V}_j$ of the j-th device's S_j, we can get

$$\begin{aligned}
\mathbf{Q}_i' &= \{\mathbf{Q}_{ik} \mid k \in [2, P]\}, \\
\mathbf{K}_i' &= \{\mathbf{K}_{ik} \mid k \in [1, P-1]\}, \\
\mathbf{V}_i' &= \{\mathbf{V}_{ik} \mid k \in [1, P-1]\}, \\
\mathbf{O}_{ij} &= \begin{cases} \text{CausalATTN}(\mathbf{Q}_i, \mathbf{K}_j, \mathbf{V}_j) & \text{if } i >= j, \\ \text{CausalATTN}(\mathbf{Q}_i', \mathbf{K}_j', \mathbf{V}_j') & \text{if } i < j. \end{cases}
\end{aligned} \tag{14}$$

By adopting a zigzag-way workload balance or striped-way workload balance, each device has exactly the same compute workload, and FlashAttention's optimization of causal attention can be further leveraged to achieve workload balance at the streaming processor level. For BurstEngine, both zigzag-way workload balance and striped-way workload balance can be integrated. From our pilot experiments, integrating BurstEngine and striped-way workload balance achieves better performance.

Block-wise Sparse Attention. In addition to causal attention, block-wise sparse attention is another common sparse attention pattern. In block-wise sparse attention, the input sequence is divided into blocks, and each token only attends to tokens within the same block or some neighboring blocks. This pattern is widely used in reducing the computation and memory consumption of training Transformers [11, 12]. While sparse patterns are inherently difficult to integrate with conventional Context Parallelism methods due to extreme workload imbalance among workers, BurstEngine adopts a strategy similar to striped-way workload balance to balance the workload for block-wise sparse attention. In block-wise sparse attention, we first divide the sequence into blocks along the sequence dimension, and the size of each block $\mathbf{N}_{\text{blk}}$ needs to be a multiple of the number of devices G, which is a strict requirement for block-wise sparse attention workload balance. Then we define the block-masking matrix as $\mathbf{M}_{\text{blk}} \in \mathbb{R}^{\frac{\mathbf{N}}{\mathbf{N}_{\text{blk}}} \times \frac{\mathbf{N}}{\mathbf{N}_{\text{blk}}}}$, where $\mathbf{M}_{\text{blk}}[i, j] = 1$ if all tokens in the i-th block can attend to the tokens in the j-th block. Then, as shown in Figure 11, we adopt a strategy similar to striped-way workload balance for block-wise sparse attention. With the strategy, each device can get exactly the same compute workload, and there is no unnecessary idle time for each device since the workload is balanced.

4 EXPERIMENTS

4.1 Experimental Settings

Hardware Settings. We adopt two configurations: 32×A800 and 64×A800. For all configurations, each node is equipped with 8×A800-SXM4-80GB GPUs linked via 400GB/s NVLink, 8 NVIDIA Mellanox HDR InfiniBand NICs(200Gb/s), and 128 CPU cores.

Model Sizes. In experiments, we mainly perform experiments on two settings of model sizes: LLaMA Transformer [9, 33, 34] with 7 billion parameters (7B, 32 layers, 32 heads, 4096 dimensions, 32K vocabulary tokens) and with 14 billion parameters (14B, 40 layers, 40 heads, 5120 dimensions, 120K vocabulary tokens).

Training Settings. We adopt fully sharded data parallelism (FSDP) [31, 42] to partition model parameters across devices. All evaluations are conducted with gradient checkpointing [4] to achieve better training efficiency under limited memory. For 7B and 14B models, we respectively set the sequence length to 2M and 1M on 32×A800, and to 4M and 2M on 64×A800.

Implementation details For BurstEngine, we adopt the FSDP implementation from BMTrain [29, 38], which achieves overlap of communication and computation at the Transformer-block level and supports optimizer offloading [32]. Our Topology-aware ring communication is built on top of NCCL and uses multi-stream programming to overlap inter-node communication, intra-node communication and computation. Additionally, we incorporate sequence-level fusion of LM head and loss fusion to reduce the memory consumption of storing the outputs of the LM head.

Evaluation Metrics. For evaluation of training efficiency, we employ tokens per second per GPU (TGS) and model FLOPs utilization (MFU). TGS provides a direct measure of end-to-end training throughput, while MFU reflects the actual utilization of the GPU device. To evaluate memory consumption, we measure the peak memory allocated on each GPU for each method, since peak memory can directly evaluate the scalability of the method to larger models or longer sequences.

Baselines. We compare BurstEngine with the following baselines in our experiments: (1) **Megatron Context Parallelism (CP)**: Megatron-LM's implementation [28] for context parallelism, using RingAttention integrated with zigzag-way workload balance. (2) **DeepSpeed-Ulysses**: DeepSpeed's implementation [18] for head parallelism. (3) **LoongTrain-DoubleRing**: LoongTrain's implementation [13] for context parallelism, using DoubleRingAttention integrated with zigzag-way workload balance. (4) **LoongTrain-USP**: LoongTrain's implementation for Head-Context Hybrid Parallelism [10, 13], achieving the state-of-the-art performance for long-sequence Transformer training, adopts head-first device placement and RingAttention with zigzag-way workload balance.

4.2 Training Performance

Throughput Performance. For comparisons of training efficiency, we evaluate the end-to-end training throughput of four

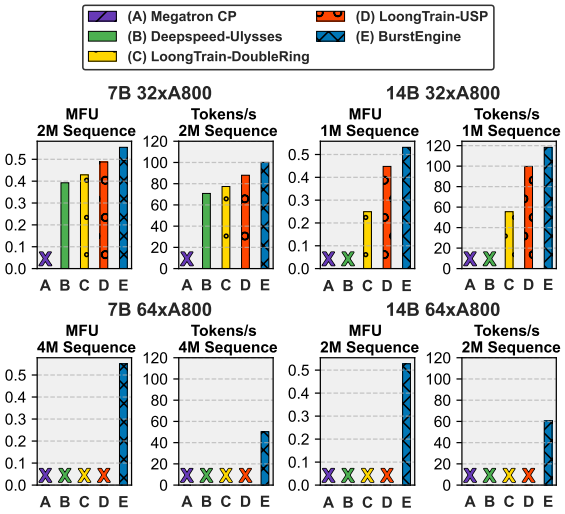

Figure 12: The end-to-end training throughput (TGS) of BurstEngine and other baselines.

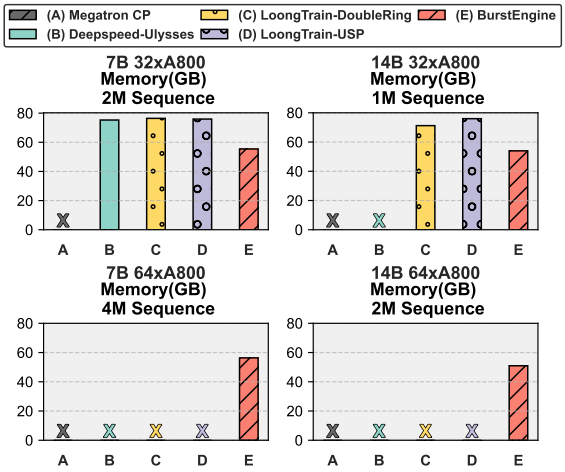

Figure 13: The peak memory usage (GB) per GPU in the end-to-end training of BurstEngine and other baselines.

baselines and BurstEngine on 7B and 14B models. As shown in Figure 12, BurstEngine outperforms all baselines in terms of TGS and MFU on both 7B and 14B models. BurstEngine achieves up to 1.19×/1.15× speedups on 7B/14B models, respectively, on 32×A800 GPUs, compared to the state-of-the-art method LoongTrain-USP. In the 7B/14B model training under the 32×A800 setting, Megatron-CP fails because of the out-of-memory issue, which is mainly due to the huge memory consumption of storing optimizer states and model weights since Megatron does not provide FSDP implementation and optimizer offloading. DeepSpeed-Ulysses has less memory consumption since it has FSDP and optimizer offloading, but still performs worse than LoongTrain-USP and BurstEngine because it can not overlap all-to-all communication with computation.

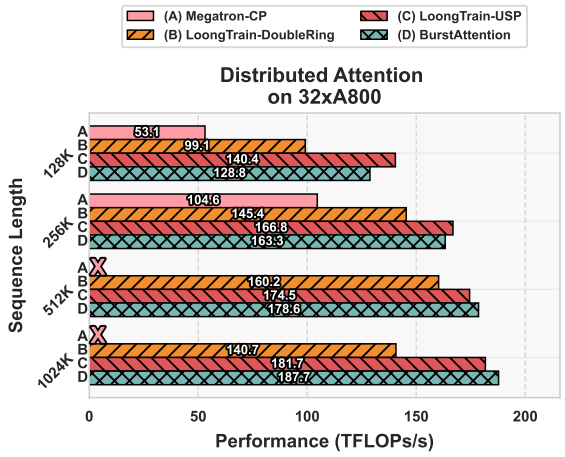

Figure 14: Performance of different distributed attention implementations

LoongTrain-DoubleRing demonstrates superior throughput performance compared to DeepSpeed-Ulysses, leveraging its advanced capability to overlap communication with computation. However, it still underperforms compared to LoongTrain-USP, primarily because of the significant communication cost that remains unoptimized. Though LoongTrain-USP has higher training throughput than other baselines, it still suffers from the communication cost of Head Parallelism and RingAttention, respectively. BurstEngine achieves the best training efficiency (TGS and MFU), showing the efficiency of backward communication optimization and topology-aware ring communication with fine-grained overlap.

Memory Performance. In terms of memory performance, as illustrated in Figure 13, BurstEngine exhibits the lowest peak memory usage. The figure highlights that DeepSpeed-Ulysses, LoongTrain-DoubleRing, and LoongTrain-USP exhibit comparable memory consumption under the setting (7B model, 32×A800), while BurstEngine saves 26.4% memory compared to the best baseline. Under the setting (14B model, 32×A800), DeepSpeed-Ulysses encounters an out-of-memory error due to its limitation on the number of model heads. LoongTrain-DoubleRing and LoongTrain-USP suffer from high memory consumption, which is mainly due to storing the outputs of the LM head. Under this setting, BurstEngine saves 24.2% memory compared to the best baseline by adopting sequence-level fusion of LM head and loss function. Under the setting of 64×A800, BurstEngine can support training a 7B model with a 4M sequence length and a 14B model with a 2M sequence length, which all baseline models fail to achieve. This is because BurstEngine exhibits nearly identical memory usage, indicating that BurstEngine achieves almost linear scaling with the device number along the sequence dimension.

4.3 Attention Performance

We evaluate the performance of BurstAttention using a 14B model's attention configuration on a setup of 32× A100 GPUs, comparing it with other RingAttention implementations such as Megatron's CP,

Table 2: The ablation study of BurstEngine for using 32×A800 to train a 14B model on the sequences of 1M tokens.

Backward Communication Optimization	Topology-aware Ring Communication	Sequence-level Fusion of LM head and Loss	Sequence-level Selective Checkpointing	Selective Checkpointing++	MFU (%)	TGS	Memory (GB)
×	×	×	×	×	36.75	83.79	48.47
✓	×	×	×	×	38.37	87.48	49.31
✓	✓	×	×	×	41.69	95.06	48.97
✓	✓	✓	×	×	41.58	94.81	**41.45**
✓	✓	✓	✓	×	47.72	108.82	45.93
✓	✓	✓	×	✓	**51.68**	**117.83**	53.91

LoongTrain's DoubleRingAttention, and USP. Given that the configuration involves 40 attention heads, DeepSpeed-Ulysses can not be applied in this scenario, as head parallelism is infeasible when the number of heads is not divisible by the number of GPUs. The experimental results, illustrated in Figure 14, highlight the efficiency of each method. From experimental results, Megatron's CP implementation encounters evaluation failures due to an out-of-memory issue when the sequence length exceeds 256k. Additionally, even in scenarios without memory issues, Megatron-CP performs poorly due to significant inter-node communication overhead. BurstAttention outperforms all other distributed attention implementations, achieving a 1.05× speedup over LoongTrain's USP and a 1.33× speedup over LoongTrain's DoubleRingAttention under 1M sequence setting. This superior performance can be attributed to BurstAttention's topology-aware ring communication strategy and its ability to finely overlap communication and computation.

At first glance, it might be a bit confusing why BurstAttention shows only marginal improvements compared to LoongTrain's USP, while BurstEngine achieves a 1.2× speedup. This is due to the difference between benchmarking attention alone and end-to-end training. When benchmarking attention alone, communication and computation can be overlapped perfectly. In end-to-end training, extra communication operations caused by FSDP result in huge communication costs and make perfectly overlapping impossible. In these cases, reducing communication cost leads to much bigger improvements in performance.

Table 3: The throughput of integrating BurstEngine with different sparse attention masking strategies.

Implementation	TGS	Speedup
Attention Masking	227.58	1.00×
Causal Attention	393.44	1.72×
SWA	837.79	3.68×

Table 4: The performance of BurstEngine across different nodes, with each node having 8×A800.

Nodes	Sequence	MFU (%)	TGS	Memory (GB)
2	0.5M	53.1	223.25	63.13
4	1M	53.2	118.36	53.96
8	2M	52.7	60.49	50.96

Table 5: The performance of BurstEngine across different context parallel size settings on 8×A800.

CP	Sequence	MFU (%)	TGS	Memory (GB)
1	32K	47.34	1201.14	57.71
2	64K	48.85	928.24	55.18
4	128K	50.55	639.43	55.58
8	256K	51.90	393.44	53.56

4.4 Ablation study

Main Optimization Strategies. We present an ablation study to assess the impact of individual optimization strategies in BurstEngine. The experiments are conducted using a 14B model and sequences of 1M tokens, evaluated on 32×A800. As shown in Table 2, we can observe how each optimization strategy contributes to the overall performance of BurstEngine. Backward communication optimization brings approximately a 1.05× speedup, while topology-aware ring communication and fine-grained communication-computation overlap contribute to an approximately 1.08× speedup. Furthermore, sequence-level fusion of LM head and loss function can save 15.3% memory compared to the baseline without introducing any performance degradation in training efficiency since there is no additional computation overhead. Sequence-level selective checkpointing can save another 14.8% memory compared to the baseline and can achieve a 1.14× speedup compared to the baseline with full checkpointing, positioning it as an optimal solution that balances memory and throughput. In the end, BurstEngine achieves a 1.4× speedup and saves at least 15% memory compared to the baseline without any optimization.

Workload Balance for Sparse Attention. We present an ablation study to assess the impact of workload balance for sparse attention in BurstEngine. The experiments are conducted using a 14B model and sequences of 1M tokens, on 32×A800. We measure the training throughput of three sparse attention implementations:

BurstEngine w. Attention Masking is the implementation of sparse attention without any workload optimization, which simply applies attention masking to restrict the attention range of each token and has the same computation overhead as full attention.

BurstEngine w. Causal Attention is the implementation of integrating BurstEngine with zigzag-way workload balance for causal attention, which can balance the workload of each device and avoid unnecessary communication and computation overhead.

BurstEngine w. SWA is the implementation of integrating BurstEngine with Sliding Window Attention (SWA). In this implementation, we adopt the block-wise sparse method, as illustrated in Figure 11, to balance the workload of each device.

As shown in Table 3, we can see that BurstEngine w. Causal Attention achieves a 1.72× speedup compared to BurstEngine w. Attention Masking. As for the SWA pattern with a 32K sliding window size, BurstEngine achieves a 3.68× speedup compared to the baseline without any workload balance strategy. While there still remains a gap in training efficiency compared to the theoretical optimization effect, BurstAttention has significantly reduced the idle occupancy caused by an imbalanced workload across devices. However, there remains potential for further optimization in communication patterns for sparse attention and single-device kernel optimization, which will be the focus of our future work.

4.5 Scalability Analysis

The scalability experiments are divided into two parts:

Intra-node scalability: In this experiment, we assess the training efficiency and memory consumption of BurstEngine using 8×A800 GPUs within a single node, evaluating its behavior under different context parallel sizes from 1 to 8. Besides, we enable optimizer offloading [32] since the memory consumption of optimizer states is quite high as the world size is small.

Inter-node scalability: We evaluate BurstEngine's training efficiency and memory consumption across different numbers of nodes from 2 to 8. Here, we disable optimizer offloading since optimizer states can be stored in each worker now. For all these settings, we set the sequence length per GPU to 32K, and the total training sequence length is $n * 32k$, in which n is the number of GPUs. As shown in Table 5, BurstEngine's MFU exceeds 50% in the single-node scenario with the context parallel size ≥ 4 and the sequence length $\geq$ 128K. In the 8×A800 setting, BurstEngine achieves a TGS of 393.44 tokens/s per GPU with a context parallel size of 8 and a sequence length of 256K. Meanwhile, BurstEngine's memory consumption remains stable as the context parallel size and the sequence length increase proportionally.

Table 4 shows BurstEngine can achieve an MFU of 52.7% with 8 nodes when processing sequences of 2M tokens. The MFU remains stable and memory consumption as the node number and the sequence length increase, showing BurstEngine can scale when extending to multiple node settings.

5 RELATED WORK

To improve the efficiency of Transformers in processing longer sequences, several optimization techniques have been proposed.

Korthikanti et al. [20] introduce selective activation recomputation, which avoids storing attention Softmax logits during the forward pass and then recomputes these logits during the backward pass. Rabe et al. [30] formalize attention modules at the block level, and assign each thread block on the device to handle the attention computation of a subsequence, further reducing temporary memory usage and achieving logarithmic memory complexity with respect to sequence length. Dao et al. [8] develop FlashAttention, a CUDA-based implementation that utilizes the high-speed I/O capabilities of SRAM to offer even greater performance improvements. While these works significantly reduce the memory consumption of attention modules for processing long sequences, they still face limitations due to the performance constraint of individual devices and bring huge computational costs as the sequence length grows.

To overcome this, some efforts have been made to utilize distributed clusters. Adopting general parallelism strategies [16, 31, 32, 35] is the most straightforward, especially using Tensor Parallelism [20, 27]. Besides, Context Parallelism methods like RingAttention [10, 13, 22, 24] and Head Parallelism [18] like DeepSpeed-Ulysses have also been proposed, which distribute the attention computation across multiple devices along the sequence dimension or head dimension. Meanwhile, to better balance the recomputation cost of FlashAttention, Li et al. [21] propose the selective checkpointing++, which stores FlashAttention's output and avoids recomputation in the backward pass. Other work, such as [25, 39] focus on optimizing the memory consumption of the LM head and propose adopting ideas similar to FlashAttention for the LM head and cross-entropy.

While these works can support efficient training when sequence length grows to 128k, 256k, or even 512k, they still suffer from high memory consumption and performance degradation when sequence length extends to 1M or even longer. Moreover, existing methods poorly integrate with sparse attention, especially as sequences grow extremely long. Thereby, we propose BurstEngine, a system that builds upon these optimizations and further reduces the communication, memory, and computation overhead through our specific optimizations.

6 CONCLUSION

BurstEngine presents a novel and efficient framework for training transformer-based models on long-sequence data. By introducing BurstAttention, sequence-level selective checkpointing, sequence-level fusion of language modeling head and loss function, and sparse attention integration, BurstEngine addresses the scalability and efficiency challenges posed by long-sequence training. These optimizations achieve a 1.2× speedup over the state-of-the-art baseline while reducing memory consumption by 26.4%. BurstEngine demonstrates the capability to support training on extremely long sequences exceeding 1M tokens, paving the way for more efficient and scalable approaches to training more effective LLMs and LMMs.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (2024YFB4505603), the National Natural Science Foundation of China (No.62192784), the high-quality development project of MIIT, and the Institute Guo Qiang at Tsinghua University. Cheng Yang is also supported by the Young Elite Scientists Sponsorship Program (No.2023QNRC001) by CAST.

REFERENCES

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219* (2024).

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

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of NeurIPS*. 1877–1901.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[5] Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, et al. 2024. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 24185–24198.

[6] Wenliang Dai, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Junqi Zhao, Weisheng Wang, Boyang Li, Pascale Fung, and Steven Hoi. 2023. InstructBLIP: towards general-purpose vision-language models with instruction tuning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*. 49250–49267.

[7] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with io-awareness. In *Proceedings of NeurIPS*. 16344–16359.

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[10] Jiarui Fang and Shangchun Zhao. 2024. A Unified Sequence Parallelism Approach for Long Context Generative AI. *arXiv preprint arXiv:2405.07719* (2024).

[11] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. Megablocks: Efficient sparse training with mixture-of-experts. *Proceedings of Machine Learning and Systems* 5 (2023), 288–304.

[12] Scott Gray, Alec Radford, and Diederik P Kingma. 2017. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224* 3, 2 (2017), 2.

[13] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, et al. 2024. Loongtrain: Efficient training of long-sequence llms with head-context parallelism. *arXiv preprint arXiv:2406.18485* (2024).

[14] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. 2022. A survey on vision transformer. *IEEE transactions on pattern analysis and machine intelligence* 45, 1 (2022), 87–110.

[15] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of NuerIPS*. 103–112.

[17] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

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

[19] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. 2022. Transformers in vision: A survey. *ACM computing surveys (CSUR)* 54, 10s (2022), 1–41.

[20] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. In *Proceedings of MLSYS*.

[21] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Xuezhe Ma, Ion Stoica, Joseph E Gonzalez, and Hao Zhang. 2024. DISTFLASHATTN: Distributed Memory-efficient Attention for Long-context LLMs Training. In *First Conference on Language Modeling*.

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

[23] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2022. A survey of transformers. *AI open* 3 (2022), 111–132.

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

[25] Cheng Luo, Jiawei Zhao, Zhuoming Chen, Beidi Chen, and Anima Anandkumar. 2024. Mini-Sequence Transformer: Optimizing Intermediate Memory for Long Sequences Training. (9 2024). [Online; accessed 2024-12-26].

[26] Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867* (2018).

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using Megatron-LM. In *Proceedings of SC*.

[28] NVIDIA. 2023. TransformerEngine. https://github.com/NVIDIA/TransformerEngine/blob/main/transformer_engine/pytorch/attention.py#L1644.

[29] OpenBMB. 2023. BMTrain: Efficient Training for Big Models. https://github.com/OpenBMB/BMTrain.

[30] Markus N Rabe and Charles Staats. 2021. Self-attention Does Not Need $O(n^2)$ Memory. *arXiv preprint arXiv:2112.05682* (2021).

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of SC*.

[32] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proceedings of ATC*. 551–564.

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

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

[35] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* (1990), 103–111.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of NeurIPS*.

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

[38] Yuzhong Wang, Xu Han, Weilin Zhao, Guoyang Zeng, Zhiyuan Liu, and Maosong Sun. 2024. H3T: Efficient Integration of Memory Optimization and Parallelism for Large-scale Transformer Training. *Advances in Neural Information Processing Systems* 36 (2024).

[39] Erik Wijmans, Brody Huval, Alexander Hertzberg, Vladlen Koltun, and Philipp Krähenbühl. [n. d.]. *Cut Your Losses in Large-Vocabulary Language Models*. arXiv:2411.09009 [cs] http://arxiv.org/abs/2411.09009

[40] Genta Indra Winata, Samuel Cahyawijaya, Zhaojiang Lin, Zihan Liu, and Pascale Fung. 2020. Lightweight and efficient end-to-end speech recognition using low-rank transformer. In *Proceedings of ICASSP*. 6144–6148.

[41] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* (2024).

[42] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. https://dl.acm.org/doi/10.14778/3611540.3611569. *Proceedings of the VLDB Endowment* 16, 12 (8 2023), 3848–3860. [Online; accessed 2024-12-18].

[43] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. 2024. MiniGPT-4: Enhancing Vision-Language Understanding with Advanced Large Language Models. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=1tZbq88f27

[44] Zhuzilin. 2023. Ring-Flash-Attention. https://github.com/zhuzilin/ring-flash-attention.git.