InternEvo: Efficient Long-Sequence Large Language Model Training via Hybrid Parallelism and Redundant Sharding

Qiaoling Chen[*1,2,3], Diandian Gu[*1,4], Guoteng Wang[1], Xun Chen[5], Yingtong Xiong[1], Ting Huang[5]
Qinghao Hu[1,2,3], Xin Jin[4], Yonggang Wen[2], Tianwei Zhang[2], Peng Sun[1,5]

[1]*Shanghai AI Laboratory* [2]*Nanyang Technological University* [3]*S-Lab, NTU*
[4]*Peking University* [5]*SenseTime Research*

Abstract

Large language models (LLMs) with long sequences begin to power more and more fundamentally new applications we use every day. Existing methods for long-sequence LLM training is neither efficient nor compatible with commonly-used training algorithms such as FlashAttention. We design InternEvo to address these issues. InternEvo decouples all of the sharding dimensions into a new hierarchical space, and systematically analyzes the memory and communication cost of LLM training. Then, it generates an effective hybrid parallelism strategy. We design a new *selective overlap* mechanism to mitigate the communication overhead introduced by the hybrid parallelism. We also implement memory management techniques to reduce GPU memory fragmentation. Evaluation results show that InternEvo generates parallelization strategies that match or outperform existing methods in model FLOPs utilization.

1 Introduction

With the emergence of large language models (LLM) in recent years, researchers have investigated and proposed many advanced training methodologies in a distributed way, such as 3D parallelism (including data parallelism [1–4], tensor parallelism [5], and pipeline parallelism [6, 7]), PyTorch FDSP [8], and automatic parallelization frameworks [9]. Recently, LLMs with long sequences have driven the development of novel applications that are essential in our daily lives, including generative AI [10] and long-context understanding [11, 12]. With the increased popularity of ChatGPT, long dialogue processing tasks have become more important for chat applications than ever [13]. In addition to these scenarios for language processing, Transformer-based giant models also achieve impressive performance in computer vision [14–16] and AI for science [17, 18], where inputs with long sequences are critical for complex tasks such as video stream processing [19] and protein property prediction [20].

Training LLMs with long sequences requires massive memory resources. To fit the memory constraint of GPUs, directly applying 3D parallelism [21, 22] or existing automatic parallelization frameworks [9, 23–26] is inefficient in training speed [27] since these system requires us to hold the whole sequence in one GPU, which limits the length of the input sequence. Several methods have been proposed to train LLMs with long sequences while not sacrificing the training speed significantly [28–31]. However, they suffer from two limitations. *First, some studies do not have satisfactory usability*. These parallelization strategies and algorithms [29–31] compute self-attention in a way that conflicts with FlashAttention [32]. FlashAttention, known for its speed and memory efficiency, is widely utilized in many LLM training [33, 34]. *Second, the solutions with good usability are still not efficient enough in training performance*. Prior methods such as DeepSpeed Ulysses introduce large inter-node communication overhead [28]; other methods like Megatron-LM largely increase the communication demand [27]. Additionally, these solutions adopt the *rigid* parallelization strategy. To quantify this, Figure 1 shows the actual materialized sizes of the memory usage of the model states and activation with the corresponding communication cost for different systems at optimal performance. To alleviate the memory pressure with a sequence length of 32K, both DeepSpeed and Megatron-LM simply partition the model states and activation at the same level (Figure 1(a)). Unfortunately, this simplicity incurs the corresponding high communication overhead (Figure 1(b)). They do not consider the differences in memory requirements and communication patterns when training with various configurations (e.g. model sizes, batch sizes, sequence lengths), making them not always optimal in different cases.

To bridge these gaps, we propose InternEvo, a parallelization framework for training Transformer-based LLMs with long sequences. InternEvo can automatically find an efficient parallelization strategy while different GPU memory management dimensions, including sequence parallelism, traditional 3D parallelism, etc. InternEvo is also compatible with efficient self-attention optimization algorithms such as FlashAttention.

However, the characteristics of long-sequence LLM training introduce two challenges for an efficient parallelization strategy. Firstly, there is a trade-off between the computation speed and communication overhead, making it difficult to find an efficient strategy. The challenge arises from the fact that reducing the computation load on each GPU by increasing the degree of GPU memory dimensions results in a proportional increase in communication demand, leading to larger communication overhead. Secondly, the default mem-

减少单个 GPU 的计算压力 = 使用更多 GPUs 进行计算，因此通信更多?

ory management method for long-sequence LLM training cannot fully utilize GPU memory, which is prone to out-of-memory (OOM) errors. This is caused by two factors. On one hand, parallelization strategies with efficient training speed might require frequent memory allocation and release, introducing memory fragmentation. On the other hand, the architecture of LLaMa-based LLMs requires memory allocation of irregular sizes [35]. This also introduces memory fragmentation and limits memory utilization.

频繁的内存分配/释放 = 中间结果存储的动态变化?

To address the first challenge, we redesign the sharding of LLMs and decouple all of the GPU memory management dimensions into a new hierarchical space with four parallel dimensions and three sharding dimensions. In this entirely new search space, we analyze the memory and communication costs of each dimension. Based on the analysis, we design an execution simulator to estimate the final memory and communication cost of every possible parallelization strategy and search for an optimal solution. Figure 1 shows that `InternEvo` performs well in storing more model states in exchange for fewer related communications. We also propose a selective overlap approach to further hide the communication overhead in computation. To address the second challenge, we design a memory pool for unified memory management to mitigate the memory fragmentation problem. This enables high memory utilization while maintaining high training efficiency.

In summary, we make the following contributions:

- We summarize the existing memory management dimensions of model training and propose a novel hierarchical space with four parallel dimensions and three sharding dimensions. We systematically analyze the memory and communication costs in this space.
- We design `InternEvo`, an automatic framework for efficient long-sequence LLM training, which automatically finds an efficient parallelization strategy.
- We design a fine-grained computation-communication overlap approach to sufficiently hide the communication overhead, and memory management techniques to reduce the GPU memory fragmentation.
- We implement a system prototype of `InternEvo`. Evaluation results on training large models with billions of parameters and up to 256k sequence length show that `InternEvo` can bring up to 4.8× model FLOPs utilization (MFU) improvement compared to competitive baselines.

2 Background

We provide a brief introduction to the essential background of Transformer-based LLM architectures and the parallelization techniques for distributed training.

2.1 Transformer Architecture

LLMs such as GPT-3 [36] and LLaMA [35] commonly embrace the Transformer architecture [37]. The architectural homogeneity suggests that tailored system optimizations are applicable. As shown in Figure 2 (a), each Transformer layer has an attention block with D attention heads and an MLP layer. Input/output dimensions are $B \times S \times H$ (B: micro-batch size, S: sequence length, H: hidden dimension). We show the memory required to store activations (ACT) in a 16-bit floating point format for each element. The fused kernel of FlashAttention [32] is used for efficient multi-head attention (MHA) computation.

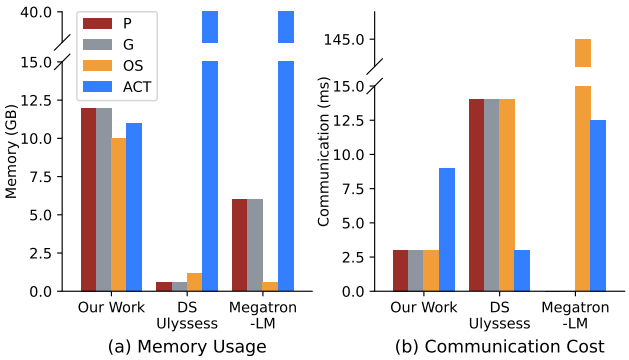

Figure 1: Memory and communication costs of model states (parameters (P), gradients (G), and optimizer states (OS)) and activations (ACT) when training a 7B model with 32k sequence length on 128 GPUs. DS means DeepSpeed here.

L	Number of Transformer Layers	b	Micro-Batch Size
V	Vocabulary Size	n	Micro-Batch Number
D	Number of Attention Heads	s_{pp}	Pipeline Parallel Size
S	Sequence Length (Tokens)	s_{tp}	Tensor Parallel Size
H	Hidden Dimension Size	s_{sp}	Sequence Parallel Size
B	Global-Batch Size (Tokens)	s_{dp}	Data Parallel Size
M	GPU Memory Usage	s_{ps}	Parameter Sharding Size
T^i	Time Consumption of Layer-i	s_{gs}	Gradient Sharding Size
N	Number of GPUs for the Job	s_{oss}	OS Sharding Size
Ψ	Parameter Count of a Layer	a	ACT Recomputation

Table 1: Notations used in this paper.

2.2 Parallelization in Distributed Training

Training LLMs efficiently at scale typically necessitates the combination of various parallelization approaches as follows.

Data Parallelism (DP) divides the input data into shards and distributes them across GPUs. Each GPU independently conducts computations to obtain gradients, which are subsequently synchronized through all-reduce.

Tensor Parallelism (TP) divides parameters along specific dimensions across GPUs to parallelize the training. As shown in Figure 2 (b), Megatron-LM leverages TP to partition linear layers along the row or column dimension, and inserts communication operations to ensure consistent results.

Pipeline Parallelism (PP) evenly divides the Transformer layers of a model into multiple stages and distributes them across GPUs. A scheduler splits an input batch into micro-batches and processes forward and backward computations alternately [6,21]. Two consecutive pipeline stages exchange intermediate data through point-to-point communication. Each pipeline stage exhibits data dependencies on others, result-

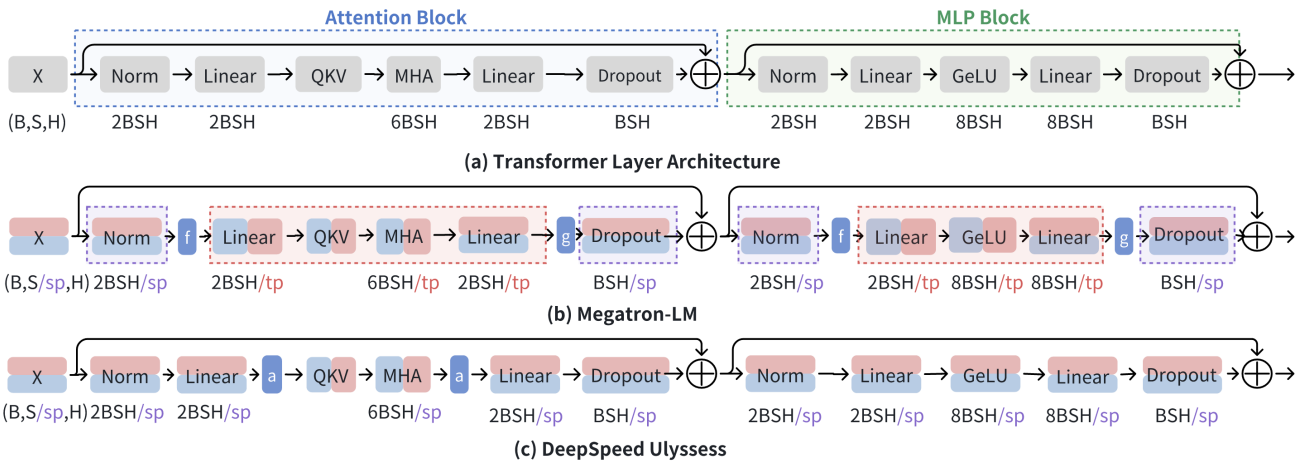

Figure 2: Background of long sequence LLM training. (**a**) Attention and MLP blocks in the Transformer architecture. The activation size (byte) of each layer is annotated. (**b**) Hybrid sequence parallelism of Megatron-LM [27]. f and g represent conjugate communication operators. In the forward pass, f: `all-gather` and g: `reduce-scatter`. In the backward pass, it is the opposite. (**c**) Sequence parallelism of DeepSpeed [28]. a represents the `all-to-all` communication operator.

Model	H	L	D	V	Data	Memory (Byte)
7B	4096	32	32	100k	P	$2(\Psi L+2VH)$
13B	5120	40	40	100k	G	$2(\Psi L+2VH)$
30B	6144	60	48	100k	OS	$6(\Psi L+2VH)$
65B	8192	80	64	100k	ACT	$2BS(17HL+H+V)$

Table 2: Model configurations and memory footprints. Ψ (parameter count of a Transformer layer) = $12H^2+2H$.

ing in computation stalls, commonly referred to as pipeline *bubbles*.

2.3 Memory Footprint Conservation

Table 2 concludes the memory footprint of different model sizes in the context of mixed precision training with the Adam optimizer [38]. With the increase of the sequence length, the memory occupied by model states constitutes only a small portion of the total memory. To alleviate the huge memory requirement caused by long sequences, some advanced system techniques are proposed:

Zero Redundancy Optimizer (ZeRO) [39]. ZeRO reduces redundant memory usage in DP by sharding model states. Specifically, ZeRO-1 partitions optimizer states (OS) across GPUs, allowing each GPU to store only one shard. ZeRO-2 further shards gradients (G), and ZeRO-3 (also implemented in FSDP [8]) further shards model parameters (P). ZeRO-1 and ZeRO-2 will not introduce extra communication. When using ZeRO-3, before conducting forward and backward computations, each GPU participates in an `all-gather` operation to collect parameters and construct complete model weights.

Activation Recomputation. It reduces activation memory consumption by checkpointing input activations in some layers, discarding intermediate activations within these layers, and recomputing them through an additional forward pass during the backward pass [40]. It efficiently reduces the activation memory. However, the recomputation introduces an approximately 30% increase in computation overhead [27].

2.4 Long-sequence Training

Processing long sequences is crucial for supporting important applications such as dealing with longer histories in chat applications. To this end, sequence parallelism (SP) has emerged as a technique aimed at alleviating activation memory footprints during the training of Transformers. In SP, the input tensor of each Transformer layer is divided along the sequence dimension, allowing for parallel computation across multiple GPUs. This segmentation, in conjunction with activation recomputation, results in a substantial reduction in activation memory requirements by a factor of s_{sp}. In this paper, we classify existing SP approaches into 3 stages, which correspond to slicing the sequence into Norm and Dropout modules, Linear modules, and MHA module. When enabled cumulatively:

SP-1: Norm and Dropout modules. As shown in Figure 2 (b), Megatron-LM capitalizes on TP to parallelize the linear layers and MHA, which are the most time-consuming components during training. Simultaneously, it employs SP on Norm and Dropout modules, effectively reducing the activation memory of these layers by a factor of s_{sp}, where $s_{sp} = s_{tp}$. To maintain consistency in computational results, it integrates necessary communications, including `all-gather` and `reduce-scatter` to transfer the activation in forward and backward passes. When the activation size increases with the sequence length, this way of communicating the activation will incur a high overhead.

SP-2: Add Linear modules. DeepSpeed Ulysses utilizes sequence parallelism on Linear, Norm, and Dropout layers, as shown in Figure 2 (c). An `all-to-all` communi-

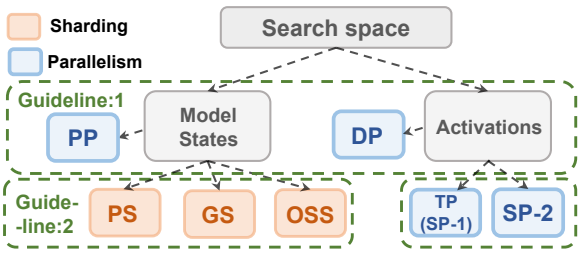

Figure 3: Search space overview. The orange and blue boxes are the dimensions in `InternEvo`'s search space.

cation is applied to the output of $Linear_{qkv}$ with a shape of $B \times 3 \times S/s_{sp} \times H$. This enables each GPU to receive the complete sequence of Q, K, and V (each with a shape of $B \times S \times H/s_{sp}$) for a subset of attention heads. Subsequently, FlashAttention is employed for highly efficient MHA computation with a constraint that the number of attention heads D should not exceed s_{sp}. Another `all-to-all` communication is then used to gather the MHA layer results and the results are re-partitioned along the sequence dimension. While DeepSpeed Ulysses achieves a reduction in activation memory usage and communication size per GPU by a factor of s_{sp} compared to Megatron-LM, it requires the complete weight matrices of all Linear layers on each GPU. Therefore, DeepSpeed Ulysses applies additional techniques such as ZeRO to reduce the memory overhead of model states.

SP-3: Add MHA modules. Existing systems like [29–31] apply sequence parallelism on all modules including MHA. In MHA, these approaches' division of Q, K, and V along the sequence dimension is coupled with a series of peer-to-peer communication operations to facilitate the cooperative attention computation. This allows for a higher parallelism dimension than the number of attention heads ($s_{sp} > D$) and enables the training of models with significantly long sequences, reaching up to 1 million tokens. However, the division in MHA is incompatible with memory-efficient mechanisms like FlashAttention, which can reduce the memory complexity from $O(S^2)$ to $O(S)$.

3 Search Space Analysis

3.1 Search Space Overview

`InternEvo` is designed to efficiently handle the memory requirements of model states and activations, along with reducing the communication overhead and improving the training performance in distributed settings. Initially, we elaborate on the rationale behind the inclusion of specific strategies in the search space and justify their relevance.

Guideline 1: Decouple the memory management strategy for model states and activations. The memory consumption of large model training can be classified into two parts: model states (encompassing parameters, gradients, and optimizer states) and activation [39]. As elucidated in Section 2.3, the memory allocation for model states represents a minor fraction of the overall memory in long-sequence training, with activation memory emerging as the primary bottleneck. The activation-reducing strategy SP-1 in Megatron-LM lacks the distinction between model states and activations during partitioning. This deficiency becomes apparent when attempting to fit activations within the memory constraints of a device, leading to a challenging dilemma. Simultaneously partitioning the sequence into smaller slices is hindered by the need to maintain constant sizes for model state slices, ensuring the communication cost of the model state remains constant. To overcome this limitation, we propose an approach that separately partitions the model states and activations. This tailored partitioning allows for independent tuning of each aspect, presenting a promising avenue for more efficient memory and communication management, especially in the context of long-sequence giant model training.

Guideline 2: Decouple memory management for model states into P, G, and OS. Despite the utilization of SP-2 by DeepSpeed, which employs ZeRO-3 to reduce the memory usage of model states by a factor of the world size n, there exists a limitation in considering the diverse memory footprints associated with the three components of model states—parameters (P), gradients (G), and optimizer states (OS), as outlined in Table 2. Based on this observation, we further decouple the memory management for model states into the management of P, G, and OS. This refined approach aims to better address the specific memory requirements of each component, potentially optimizing memory utilization in the context of long-sequence training.

这里局限性是什么没说

Guideline 3: Optimize communication by trading partial redundancy. A notable performance gap exists between intra- and inter-node network bandwidth and latency [41], posing constraints on training efficiency. In contrast to memory management for model states with zero redundancy, strategically grouping GPUs with slight memory redundancy for P, G, and OS can effectively reduce communication costs. This reduction is achieved by reducing both the size and frequency of expensive inter-node communication. A relevant example is MiCS [42], which introduces a cluster grouping strategy, partitioning all model states within each group and then replicating them across different groups.

Considering **Guidelines 1** $\sim$ **2**, we categorize GPU memory management dimensions into two main categories: model states and activations, as illustrated in Figure 3. Model state memory is governed by the pipeline parallelism (PP) strategy and three sharding strategies: parameter sharding (PS), gradient sharding (GS), and optimizer states sharding (OSS). Activation memory, on the other hand, is managed by data parallelism (DP), tensor parallelism (TP), and sequence parallelism (including SP-1 and SP-2). Overall, `InternEvo` define 4 parallel dimensions and 3 sharding dimensions as shown in the blue and orange blocks of Figure 3, and presented as $\mathbb{S} = [b, s, a, s_{pp}, s_{dp}, s_{tp}, s_{sp}, s_{ps}, s_{gs}, s_{oss}]$. Based on **Guideline 3**, different values of s_i exert a significant influence on communication costs. We find that the expensive inter-node

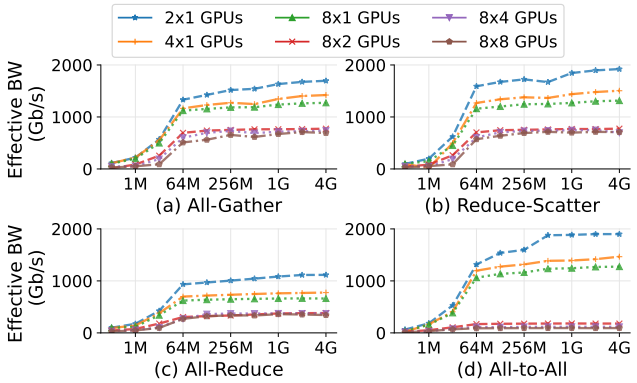

Figure 4: Performance evaluation of collective communication operations using NCCL on Nvdia Ampere GPUs. For configurations with up to 8 GPUs, the tests were executed within a single node. For scenarios with more than 8 GPUs, we extended the evaluation across nodes interconnected by 4x Mellanox Infiniband HDR NICs.

communication of certain i is could be reduced by optimizing s_i to achieve partially redundant storage. This strategic adjustment enables the consolidation of communication for i within the same node with higher bandwidth. Next, we introduce how `InternEvo` models the memory and communication cost of model states and activations for each item in $\mathbb{S}$.

3.2 Memory & Communication Analysis

We explore the memory footprint and communication costs of model training, considering different combinations of parallelism and sharding strategies. Table 1 shows notations used in this analysis. Our focus here is the costs of the L Transformer layers of a model. The embedding layer and head layer of the model will be discussed in the subsequent section.

`InternEvo` optimizes the memory usage by storing activations at Transformer layer boundaries and recomputing the remaining activations in the backward pass. When $a = 1$, `InternEvo` requires the storage of $2bSH$ bytes for activations of a Transformer layer, applicable in scenarios without pipeline or sequence parallelism. Conversely, if $a = 1$, `InternEvo` utilizes $34bSH$ bytes for activation storage. Furthermore, to enhance the efficiency of multi-head attention computation, `InternEvo` uses the fused kernel of FlashAttention by default, which already incorporates the *selective activation recomputation* technique to minimize the memory usage during attention computation [33]. The total GPU memory consumed by activations across all L Transformer layers is given by:

$$M_{ACT}(s_{sp}=1, s_{pp}=1) = \sum_{i=0}^{L-1}(2bSH + 32bSH(1-a)).$$

3.2.1 Basic Communication Cost Model

`InternEvo` uses a device mesh to represent the communication pattern [9, 26, 43]. This logical view, a 2-dimensional abstraction of physical devices, illustrates their ability to communicate along the first and second dimensions, each with distinct bandwidths. To specify the communication characteristics of participant GPUs (p), we employ a superscript notation. Specifically, p^0 denotes participant GPUs communicating along the 0-th axis via intra-node networks like NVLink, while p^1 signifies communication along the 1-th axis through inter-node networks like Infiniband. `InternEvo` uses the cost estimation formula $\tau(o, v, p^{\{0,1\}}) = v/w(o, v, p^{\{0,1\}})$ to assess the time consumption of collective communication operator (o) with a given data size (v) and a specified participant GPU count (p). Here, $w(o, v, p^{\{0,1\}})$ represents the effective bandwidth obtained through performance profiling on the target GPU cluster, as illustrated in Figure 4. Our focus in this study revolves around four key collective communication operations: `all-reduce`, `all-gather`, `reduce-scatter`, and `all-to-all`. `InternEvo` has the flexibility to extend its cost estimation to other operators as needed.

3.2.2 Analysis of Parallelism Strategies

We conduct the analysis of the memory footprint and communication cost when `InternEvo` adjusts parallelism strategies using s_{dp}, s_{pp}, s_{tp}, and s_{sp}.

Data Parallelism & Gradient Accumulation. `InternEvo` uses data parallelism to distribute data samples of a single step across s_{dp} GPU worker groups, with each group maintaining a replica of the model. Additionally, `InternEvo` facilitates the emulation of a larger batch size by cumulatively aggregating gradients from n micro-batches before executing the weight update. These combined strategies prove beneficial in alleviating the activation memory footprint M_{ACT}, which escalates linearly with the micro-batch size b. It is important to note that data parallelism introduces a necessary communication cost for synchronizing gradients, and we will analyze such costs in the context of the optimizer states sharding strategy.

Pipeline Parallelism. `InternEvo` can harness pipeline parallelism to evenly partition L Transformer layers into s_{pp} stages and distribute them across GPUs, effectively reducing the memory footprint of P, G, and OS by a factor of s_{pp}. By default, `InternEvo` employs the 1F1B scheduler, which divides an input batch into n micro-batches and processes the forward and backward computations alternately [6, 21]. Consequently, the first stage must store activations for as many micro-batches as the degree of pipeline parallelism, while subsequent stages gradually store fewer activations [21]. In case of $s_{pp} > 1$, we define the maximum GPU memory usage as follows:

$$M_{P,G,OS}(s_{pp} \geq 1) = M_{P,G,OS}(s_{pp} = 1)/s_{pp},$$
$$M_{ACT}(s_{pp} \geq 1) = \min(s_{pp}, n) \times M_{ACT}(s_{pp} = 1)/s_{pp}.$$

In the pipeline parallelism setup, consecutive stages exchange intermediate data through point-to-point (P2P) communication. We exclude the consideration of such communication

costs in this work for two reasons: 1) P2P communications occur between stages rather than within each Transformer layer, resulting in a much smaller communication size compared to other aspects; 2) such communication can be overlapped with computation during model training [44]. Additionally, each pipeline stage exhibits data dependencies on others, leading to computation stalls, commonly known as a "pipeline *bubble*," which will be discussed in the subsequent section.

Tensor Parallelism. When $s_{tp} > 1$, `InternEvo` incorporates tensor parallelism to parallelize the computationally-intensive linear layers and Multi-Head Attention. Simultaneously, it defaults to utilizing sequence parallelism for normalization and dropout modules, resulting in $s_{sp} = s_{tp} > 1$. In this scenario, `InternEvo` effectively reduces the GPU memory usage of P, G, OS, and ACT by a factor of s_{tp}, as expressed by the equation:

$$M_{P,G,OS,ACT}(s_{tp} \geq 1) = M_{P,G,OS,ACT}(s_{sp} = s_{tp} = 1)/s_{tp}.$$

As shown in Figure 2, `InternEvo` uses one `all-gather` and one `reduce-scatter` operation on each Transformer layer per micro-batch during the forward pass when $s_{sp} = s_{tp} > 1$. In the backward pass, the insertion of two `all-gather` and one `reduce-scatter` communication operations occur per micro-batch. The communication time for the i-th layer of a step, introduced by tensor parallelism, is as follows:

$$\begin{aligned} T^i_{comm}(s_{tp} = s_{sp} > 1) =& 2n\tau(\texttt{reduce-scatter}, 2bSH, s_{tp}) + \\ & 3n\tau(\texttt{all-gather}, 2bSH, s_{tp}). \end{aligned}$$

Sequence Parallelism. `InternEvo` employs sequence parallelism for all modules except MHA when $(s_{sp} > 1, s_{tp} = 1)$. As illustrated in Figure 2 (c), `InternEvo` re-partitions through `all-to-all` before and after Multi-Head Attention, ensuring the hidden dimension of this module slice aligns with the sequence, similar to other modules. Consequently, `InternEvo` reduces the memory footprint of ACT by a factor of s_{sp}, as indicated by the equation:

$$M_{ACT}(s_{sp} \geq 1, s_{tp} = 1) = M_{ACT}(s_{sp} = s_{tp} = 1)/s_{sp}.$$

In each Transformer layer during both forward and backward passes, `InternEvo` incorporates two `all-to-all` communication operations per micro-batch. The initial `all-to-all` exchanges $6bSH$ bytes of data, followed by a second exchange of $2bSH$ bytes of data. The communication time attributable to sequence parallelism for the i-th layer of a step is:

$$\begin{aligned} T^i_{comm}(s_{sp} > 1, s_{tp} = 1) =& 2n\tau(\texttt{all\_to\_all}, 6bSH, s_{sp}) + \\ & 2n\tau(\texttt{all\_to\_all}, 2bSH, s_{sp}). \end{aligned}$$

It is noteworthy that this strategy necessitates the complete weight matrices on each GPU for computation.

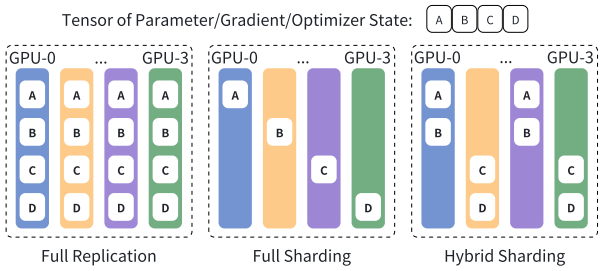

Figure 5: Three Sharding strategies.

3.2.3 Analysis of Sharding Strategies

`InternEvo` uses a mix of parallelism strategies to distribute Transformer layers and modules efficiently across GPUs. Parameters, gradients, and optimizer states may be replicated or sharded among GPUs based on two configurations:

- Config-1: $s_{dp} \geq 1, s_{pp} \geq 1, s_{tp} = s_{sp} > 1$. GPUs with the same tensor and pipeline parallelism rank hold replicated parameters, ensuring synchronized training.
- Config-2: $s_{dp} \geq 1, s_{pp} \geq 1, s_{tp} = 1, s_{sp} \geq 1$. GPUs within the same pipeline stage jointly hold replicated parameters along with their gradients and optimizer states.

As illustrated in Figure 5, `InternEvo` incorporates three sharding strategies, impacting memory footprint and communication cost. The sharding factor F denotes the number of GPUs over which the tensor is sharded. For a tensor with E elements and R GPUs, the three strategies could be outlined as follows:

- Full Replication ($F = 1$): A complete replica of the tensor with E elements is stored on each of the R GPUs.
- Full Sharding ($F = R$): Each GPU stores only E/R elements, and the full tensor can be reconstructed by performing an `all-gather` across all G GPUs.
- Hybrid Sharding ($1 < F < R$): This strategy combines sharding and replication. Each GPU holds E/F elements, and each element is replicated on R/F GPUs. The full tensor can be reconstructed by employing an `all-gather` across the F GPUs.

Unlike ZeRO-1, ZeRO-2 [39], MiCS [42], ZeRO-3 [8] and ZeRO++ [41], which use static sharding strategies for parameters, gradients, and optimizer states, `InternEvo` takes a more flexible approach: it applies three separate sharding strategies — determined by distinct factors s_{ps}, s_{gs}, and s_{oss} — to parameters, gradients, and optimizer states. This flexibility allows for a balance between communication and memory, offering tailored performance optimization during training.

Parameter Sharding. When parameters are replicated across GPUs, `InternEvo` offers the capability to shard these parameters with a sharding factor s_{ps}. This strategy, illustrated in Figure 6, effectively reduces GPU memory footprint of parameters by a factor of s_{ps}, as expressed by the equation:

$$M_P(s_{ps} \geq 1) = M_P(s_{ps} = 1)/s_{ps}.$$

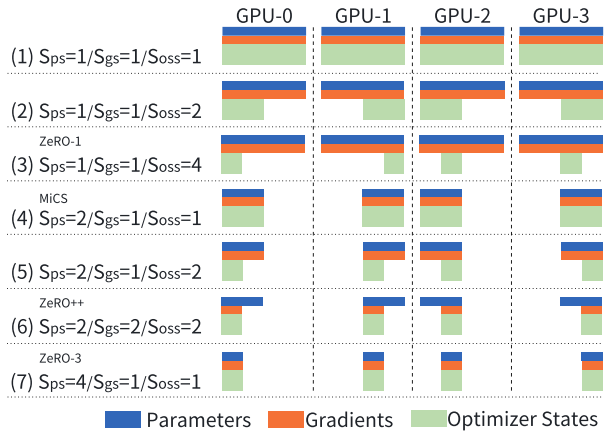

Figure 6: Overview of parameter sharding, gradient sharding, and optimizer state sharding with various sharding factors.

During both the forward and backward passes of each micro-batch, `InternEvo` orchestrates the collection of parameter shards from other GPUs to reconstruct the complete set of model weights necessary for computations. In the forward pass, a single `all-gather` operation on the corresponding parameter sharding group is employed. Subsequently, during the backward pass, each GPU computes gradients for the entire set of model parameters collected by another `all-gather` operation. Following this, `InternEvo` utilizes the `reduce-scatter` function to aggregate and distribute gradients across the parameter sharding group. The communication time attributable to parameter sharding for the i-th layer of a step is:

$$T^i_{comm}(s_{ps} > 1) = 2n\tau(\texttt{all-gather}, 2\Psi/s_{tp}, s_{ps}) + n\tau(\texttt{reduce-scatter}, 2\Psi/s_{tp}, s_{ps}),$$

where Ψ/s_{tp} is the parameter count of a Transformer layer partitioned by tensor parallelism. Additionally, the sharding factor s_{ps} is constrained by:

$$s_{ps} \in [1, N/(s_{pp} \cdot s_{tp})] \cap \mathbb{Z}.$$

This constraint ensures the feasibility of parameter sharding, considering various configurations. For example, when $s_{tp} = s_{sp} > 1, N/(s_{pp} \cdot s_{tp})$ equates to s_{dp}. When $s_{tp} = 1$ and $s_{sp} > 1$, the maximum of s_{ps} could be higher than s_{dp}, allowing for more flexibility in optimizing s_{ps}.

Optimizer State Sharding. When parameters are replicated across R GPUs and subjected to the parameter sharding operation with a factor s_{ps}, a parameter may still be distributed among multiple GPUs. For instance, in Figure 6(6), with $s_{ps} = 2$, GPU-0 and GPU-2 store the same parameters. Following the MiCS approach [42], the gradients and optimizer states associated with these parameters are also replicated on both GPU-0 and GPU-1. In contrast, `InternEvo` offers enhanced flexibility by allowing the sharding of gradients and optimizer states independently using factors s_{gs} and s_{oss}. This decoupling of sharding factors for parameters, gradients, and optimizer states provides finer control over the distribution strategy, enabling more efficient memory utilization and communication patterns during training. When $s_{oss} > 1$, the memory footprint of optimizer states can be further reduced, as expressed by the following equation:

$$M_{OS}(s_{oss} \geq 1) = M_{OS}(s_{ps} = 1, s_{oss} = 1)/(s_{ps} \cdot s_{oss}),$$

subject to the constraint:

$$s_{oss} \in [1, N/(s_{pp} \cdot s_{tp} \cdot s_{ps})] \cap \mathbb{Z}.$$

This formulation ensures that the sharding factor s_{oss} remains within a valid range, aligning with the overall parallelism and sharding strategy for optimizer states.

After the last micro-batch's backward pass, each GPU is responsible for updating the weights of parameters based on the optimizer states. Prior to the parameter update stage, each GPU is required to collect and aggregate gradients for parameters within its optimizer states. To streamline this process, `InternEvo` employs one `all-reduce` operation on gradients across GPUs that share the same set of parameters. For instance, in Figure 6(2), `InternEvo` conducts `all-reduce` on all four GPUs to synchronize the gradients. Subsequently, GPU-0 and GPU-2 select the first half of the aggregated gradients for weight updates. In Figure 6(6), `InternEvo` conducts `all-reduce` specifically on GPU-0 and GPU-2, obtaining the aggregated gradients necessary for weight updates. Following the optimizer update stage, `InternEvo` employs one `all-gather` operation on updated parameters across GPUs within the same optimizer state sharding group. This ensures that each GPU receives all the updated values for parameters stored in its local memory. For instance, in Figure 6(2), `InternEvo` conducts `all-gather` on GPU-0 and GPU-1. The communication time associated with optimizer states for the i-th layer is articulated as follows:

$$T^i_{comm}(s_{oss} \geq 1) = \tau(\texttt{reduce-scatter}, 2\Psi/(s_{tp} \cdot s_{ps}), s_{oss}) + \tau(\texttt{all-reduce}, 2\Psi/(s_{tp} \cdot s_{ps}), N/(s_{pp} \cdot s_{tp} \cdot s_{ps})).$$

Gradient Sharding. When $s_{ps} > 1$, the gradient is inherently sharded with parameters. If there are still replicas of the gradient in the cluster, `InternEvo` support gradient sharding with factor s_{gs}. This is done to effectively mitigate memory usage for gradients, particularly in the context of gradient accumulation. The memory footprint of gradients is expressed as:

$$M_G(s_{gs} \geq 1) = M_G(s_{ps} = 1, s_{gs} = 1)/(s_{ps} \cdot s_{gs}).$$

For simplicity, we restrict the choice of s_{gs} in this work:

$$s_{gs} \in \{1, s_{oss}\}.$$

For example, in Figure 6(6), gradients are sharded with $s_{gs} = 2$. To facilitate gradient accumulation, `InternEvo` operates

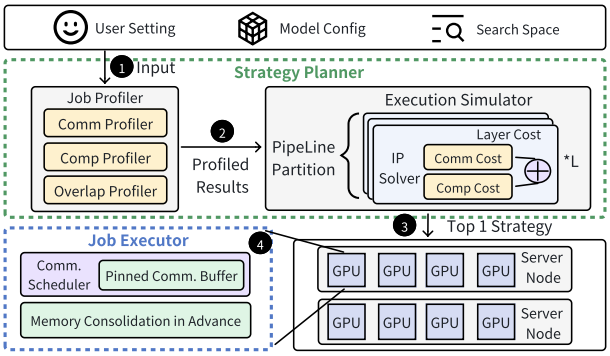

Figure 7: Overview of `InternEvo` architecture and workflow.

`all-reduce` on each micro-batch, excluding the last one, to aggregate gradients across GPUs with the same portion of parameters. The associated communication cost is given by:

$$T^i_{comm}(s_{gs}>1)=(n-1)\tau(\texttt{all-reduce},\frac{2\Psi}{s_{tp}\cdot s_{ps}},\frac{N}{s_{pp}\cdot s_{tp}\cdot s_{ps}}).$$

3.2.4 Summary

In the training of transformer-based models with a combination of parallelism and sharding strategies, we provide a comprehensive analysis of communication and memory considerations. Specifically, the GPU memory usage of all transformer layers, denoted as M_{tf}, can be articulated as follows:

$$M_{tf}=\underbrace{\frac{2L\Psi}{s_{pp}\cdot s_{tp}}(\frac{1}{s_{ps}}+\frac{1}{s_{gs}}+\frac{1}{s_{oss}})}_{\text{Memory used by P, G and OS}}+\underbrace{\frac{bSHL(34-32a))}{s_{pp}\cdot s_{sp}}}_{\text{Memory used by ACT}}. \quad (1)$$

Note that we focus on the scenario where the pipeline parallelism size exceeds the micro-batch number here ($s_{pp}>n$). Additionally, the communication time of i-th layer during a step, denoted as T^i_{comm}, can be expressed as:

$$T^i_{comm}=\underbrace{T^i_{comm}(s_{tp},s_{sp})}_{\text{Communicate ACT}}+\underbrace{T^i_{comm}(s_{ps},s_{gs},s_{oss})}_{\text{Communicate P, G}}. \quad (2)$$

Communication may be overlapped with computation, which will be explored in subsequent discussions.

4 System Design

In this section, we delve into how `InternEvo` navigates the search space, which comprises micro-batch size, micro-batch number, activation recomputation status, four parallelism strategies and three sharding strategies, denoted as $\mathbb{S}=[b,n,a,s_{pp},s_{dp},s_{tp},s_{sp},s_{ps},s_{gs},s_{oss}]$. We focus on how `InternEvo` systematically optimizes the overall training performance within this defined space.

4.1 System Architecture & WorkFlow

Illustrated in Figure 7, `InternEvo` comprises two key components: Strategy Planner and Job Executor. (1) The Strategy Planner is dedicated to identifying the optimal strategy $\mathbb{S}$ for parallelizing model training. This involves two core modules: firstly, the Job Profiler, which meticulously profiles the submitted workload to extract crucial execution information; and secondly, the Execution Simulator, tasked with simulating various strategies within $\mathbb{S}$ to pinpoint the most effective solution. (2) The Job Executor is responsible for the actual execution of the training job with the given training strategy $\mathbb{S}$ on GPU nodes. This component integrates two essential modules to enhance training performance. The Communication Scheduler, through its ability to overlap computation and communication at a fine granularity, significantly mitigates communication overhead. Simultaneously, the Fragmentation Memory Manager ensures unified GPU memory management, effectively reducing the occurrence of OOM errors attributed to GPU memory fragmentation, particularly in the context of long-sequence training scenarios.

Workflow. ❶ `InternEvo` commences with users specifying transformer model architecture like attention head number and layer number, hyper-parameters such as sequence length and global batch size, and training cluster settings like total GPU number and memory capacity. ❷ The Job Profiler conducts a comprehensive analysis by profiling the effective bandwidth of used collective communication operations, computation times for modules within the model, and identifying instances of communication-computation overlap (Section 4.2). The gained profiling results drive the Execution Simulator to estimate the memory cost and step time for various combinations of parallelism and sharding strategies (Section 4.5). Leveraging an optimization problem solver, `InternEvo` identifies the *top-10* strategies, and real system profiling is employed to determine the best one ❸. This exploration leads to the identification of a parallelization strategy with the shortest step time, ensuring the absence of out-of-memory errors. ❹ Then, the Job Executor runs the training job utilizing the corresponding strategy with high training speed and memory utilization.

4.2 Job Profiler

The Job Profiler consists of three parts: (1)`VPro` profiles the effective bandwidths for NCCL communication offline. (2) `UPro` conducts detailed profiling of a single forward pass of a transformer layer online. (3) `OPro` profiles the overlap between computation and communication offline. Details of the three components are as follows:

Communication Profiler (`VPro`). For an execution plan $\mathbb{S}=[b,n,a,s_{pp},s_{dp},s_{tp},s_{sp},s_{ps},s_{gs},s_{oss}]$, `InternEvo` leverages real-system profiling to determine the effective bandwidth of four used collective communication operations (i.e. `all-reduce`, `all-gather`, `reduce-scatter`, and `all-to-all`) across various communication sizes and device meshes. Figure 4 gives an example of the profiling results. Thus, `InternEvo` estimates the communication latency induced by parallelism and sharding strategies according to Equation 2: $T^i_{comm}=\texttt{VPro}(\mathbb{S})$.

Computation Profiler (UPro). InternEvo conducts computation time profiling for i-th transformer layer with the specified execution plan $\mathbb{S}$: $T^i_{comp} = \texttt{UPro}(\mathbb{S})$. As discussed in Section 2.1, InternEvo identifies the self-attention and linear modules as the key contributors to computation time in the Attention and MLP block. For the self-attention module, InternEvo profiles the execution time of `self-atten`$(b, S, H, D/s_{sp})$ on a single GPU using FlashAttention. Regarding the linear modules, if $s_{sp} > 1$ and $s_{tp} = 1$, InternEvo profiles the execution time of applying the input tensor with dimension $(b, S/s_{sp}, H)$ on the Linear module. In the scenario where $s_{tp} > 1$, InternEvo profiles the computation time of each linear module based on its row-partitioning or column-partition strategy, as discussed in Section 3.

Overlap Profiler (OPro). InternEvo supports the concurrent execution of communication operations alongside computation kernels. Notably, in the forward pass, it enables the simultaneous execution of computation and the `all-gather` communication resulting from parameter sharding. In the backward computation phase, InternEvo extends this capability to various communication tasks, such as `all-gather` and `reduce-scatter` induced by tensor parallelism and parameter sharding, as well as `all-reduce` from gradient sharding and optimizer states sharding. Despite the widespread practice of estimating total training time by taking the maximum value between profiled communication time (VPro) and computation time (UPro), this approach may lack accuracy. Communication primitives, such as NCCL [45], introduce a slowdown in both computation and communication due to resource contention in GPU streaming multiprocessors [24]. This contention varies with model sizes and sequence lengths, leading to performance degradation. InternEvo defines a slow down ratio $\mathcal{R}$ for OPro to tackle this challenge:

$$\texttt{OPro}(\mathbb{S}) = \mathcal{R} \cdot \max(\texttt{VPro}(\mathbb{S}), \texttt{UPro}(\mathbb{S})).$$

In this study, the parameter $\mathcal{R}$ varies within the range of 1.25 to 1.35 as determined through evaluations.

4.3 Communication Scheduler

The Communication Scheduler module in InternEvo strategically orchestrates communication and computation to enhance overall system performance. Specifically, our approach focuses on optimizing the overlap during the backward pass by implementing a refined intra-layer strategy. This involves leveraging parameter pre-fetching (`all-gather`) and gradient synchronization (`reduce-scatter`) at the granularity of modules, such as the linear module. In the forward pass, we employ an inter-layer overlap strategy to mitigate latency by pre-fetching the entire set of parameters for upcoming layers while concurrently computing the current layer. This is particularly beneficial when $s_{ps} > 1$. The iterative selective overlap approach adeptly manages both communication overhead and computation execution time, resulting in a noteworthy enhancement in overall system performance. For more design and implementation detail refers to Appendix A.1.

4.4 Memory Fragmentation Manager

The `cudaMalloc` and `cudaFree` interfaces cause substantial latency and impose significant synchronization overhead across CUDA streams. To mitigate these challenges, memory pools, such as CNMeM [46] and the PyTorch memory allocator [2], serve as an intermediary layer to handle memory allocation and deallocation requests during model training. These memory pools proactively pre-allocate contiguous memory chunks of varying sizes and employ strategies like merging or splitting chunks to meet diverse memory requirements.

Unfortunately, scenarios involving frequent (de)allocation requests can lead to memory fragmentation, posing challenges to the efficiency of memory pools and potentially impacting the training of models during peak memory usage. This issue becomes more pronounced in the context of long sequence training with communication/computation overlapping, as this case necessitates the temporary allocation of large contiguous memory chunks. In response to this challenge, we propose two measures: *Pinned Communication Buffer* and *Memory Consolidation in Advance*. Specifically, the pinned communication buffer introduces dedicated memory pools for `all-gather` communications, implementing double buffer rotation to efficiently manage memory during forward and backward passes. The memory consolidation approach proactively consolidates small memory chunks and addresses challenges posed by fragmented memory space. For more design and implementation detail refers to Appendix A.2.

4.5 Execution Simulator

The execution simulator estimates training performance using a given execution plan $\mathbb{S}$ and obtains the best solution candidates by solving an integer programming problem.

We formulate $T_{fwd\_bwd}(\mathbb{S})$, representing the total time for all forward and backward passes in a training step with the given execution plan. The equation is given by:

$$T_{fwd\_bwd}(\mathbb{S}) = (n + s_{pp} - 1) \cdot (T_{other}(\mathbb{S}) + \lceil L/s_{pp} \rceil \cdot \texttt{OPro}(\mathbb{S})),$$

Here, T_{other} is the execution time of other layers, such as the head layer and the embedding layer, which can be estimated through profiling. The execution time of a single micro-batch is given by $T_{other}(\mathbb{S}) + \lceil L/s_{pp} \rceil \cdot \texttt{OPro}(\mathbb{S})$. In the presence of a pipeline *bubble* caused by data dependencies, the total time of all forward and backward passes is obtained by multiplying the single micro-batch time with $(n + s_{pp} - 1)$.

Building upon equation 1, which defines the GPU memory usage of a transformer layer, the total GPU memory usage $M(\mathbb{S})$ can be expressed as:

$$M(\mathbb{S}) = M_{tf}(\mathbb{S}) + M_{other}(\mathbb{S}),$$

where M_{other} represents the memory used by head and embedding layers, as well as temporary buffers for communication and computation.

InternEvo formulates an optimization problem to search for optimal execution plans by minimizing the sum of the execution time of forward-backward passes and the parameter update time, subject to various constraints. The integer programming problem is defined as follows:

$$\text{Minimize} \quad T_{fwd\_bwd}(\mathbb{S}) + T_{update}(\mathbb{S}) \tag{3}$$

$$\text{Subject to} \quad M(\mathbb{S}) \leq \texttt{GPU\_Memory\_Capacity} \tag{4}$$

$$b \cdot S \cdot n \cdot s_{dp} = B \tag{5}$$

$$s_{dp} \cdot s_{sp} \cdot s_{pp} = N \tag{6}$$

$$s_{ps} \cdot s_{pp} \cdot s_{tp} \cdot i = N \quad i \in \mathbb{Z} \tag{7}$$

$$s_{ps} \cdot s_{oss} \cdot s_{pp} \cdot s_{tp} \cdot j = N \quad j \in \mathbb{Z} \tag{8}$$

$$s_{gs} \in \{1, s_{oss}\} \tag{9}$$

This problem optimizes the execution plan with respect to GPU memory capacity (Equation 4), global batch size (Equation 5), global GPU number (Equation 6), and other considerations outlined in Section 3.

For each s in $\mathbb{S}$, allocating specific tensors on the device mesh introduces multiple possible configurations (3.2.1), influencing the communication latency. Instead of exhaustively iterating through all potential GPU allocations for $\mathbb{S}$, we leverage practical insights to streamline the exploration of assignment strategies more efficiently.

Takeaway 1: In cases where $s_{sp} \geq 1, s_{pp} \geq 1, s_{dp} \geq 1$, prioritize allocating ranks of the same tensor/sequence parallelism group with the fewest physical nodes. When activation memory usage significantly exceeds model state memory, communication to activation incurs substantial overhead. Therefore, give preference to utilizing high-performance intra-node bandwidth for activation communication.

Takeaway 2: If $s_{ps} \geq 1, s_{gs} \geq 1, s_{oss} \geq 1$, prioritize allocating ranks of the same parameter sharding group with the fewest physical nodes. When $s_{ps} > 1$, frequent `all-gather` communication can impact overall training performance. Therefore, favor the use of high-performance intra-node bandwidth for parameter pre-fetching.

Following the outlined takeaways, with a given execution plan $\mathbb{S}$, we can estimate the communication process group allocations for each parallelism strategy and sharding approach. Subsequently, this allows us to generate an executable for precise profiling using `UPro` with the estimated device meshes. In Appendix A.3, we provide a detailed workflow for solving the optimization problem presented. Note that InternEvo identifies the *top-10* strategies, and the selection of the optimal strategy is finalized through real system profiling to ensure the most effective performance.

5 Evaluation

5.1 Experiment Setup

Implementation. InternEvo has been implemented using Python and encompasses around 35,000 lines of code (LOC). The codebase consists of 5,000 LOC for the Strategy Planner and 30,000 LOC for the Job Executor. For access to the system, please visit *https://github.com/InternLM/InternEvo*. Additional insights into the implementation specifics of the Communication Scheduler and Memory Fragmentation Manager can be explored in Appendix A.1 and A.2.

Testbed. InternEvo undergoes a thorough evaluation in the training of Transformer-based models using the LLaMA architecture. The models encompass parameters ranging from 7 billion to 65 billion, and detailed configurations can be found in Table 2. The training process occurs on a physical cluster comprising 16 GPU servers. Each server is equipped with 8 GPUs and 128 CPU cores, leading to a total of 128 NVIDIA Ampere GPUs. Each GPU boasts 80GB of memory. The GPUs are interconnected through NVLink and NVSwitch, while inter-node communication is facilitated by four NVIDIA Mellanox 200Gbps HDR InfiniBand.

Baselines and Metrics. We benchmark InternEvo against two state-of-the-art systems for long-sequence LLM training, namely DeepSpeed Ulysses [28] and Megatron-LM [27]. The evaluation metric is Model FLOPs Utilization (MFU) [47]. We also assess tokens per GPU per second (TGS) during training. While InternEvo automatically searches for a parallelization strategy, we manually set the number of partitions for DeepSpeed Ulysses and Megatron-LM. The chosen strategy for these baselines is the fastest one that avoids triggering an out-of-memory error for fair comparison. It is worth noting that, in practice, the parallelization strategy selected by developers for DeepSpeed Ulysses and Megatron-LM would likely be either the same as or slower than the strategy chosen in our experiments. As a comprehensive case study, we also present the parallelization strategy identified by InternEvo.

5.2 End-to-End Evaluation

We initiate our evaluation by assessing the training speed improvement achieved by the parallelization strategy identified by InternEvo while ensuring no out-of-memory errors. Figure 8 shows MFU when training models of varying sizes with different sequence lengths and Figure 10 shows the corresponding peak memory usage. For each model, we adopt a global batch size of $4M$ tokens [48, 49]. In Megatron-LM and DeepSpeed Ulysses, we first tune the best micro-batch size (b) for each model and system configuration that maximizes the system performance. The gradients are accumulated across microbatches with micro-batch numbers (n). More details of the parallelism configuration we set for Megatron-LM and DeepSpeed Ulysses can be found in Appendix A.4, and the related TGS result is shown in Figure 9.

When the sequence is short, the primary bottleneck in DeepSpeed Ulysses is the communication time. This is attributed to the utilization of ZeRO3 to reduce the memory requirements of model states, introducing a constant and substantial inter-node communication overhead. Additionally, the `all-to-all` communication overhead increases with the sequence length. Consequently, for shorter sequences ($S \leq 32K$), MFU tends to

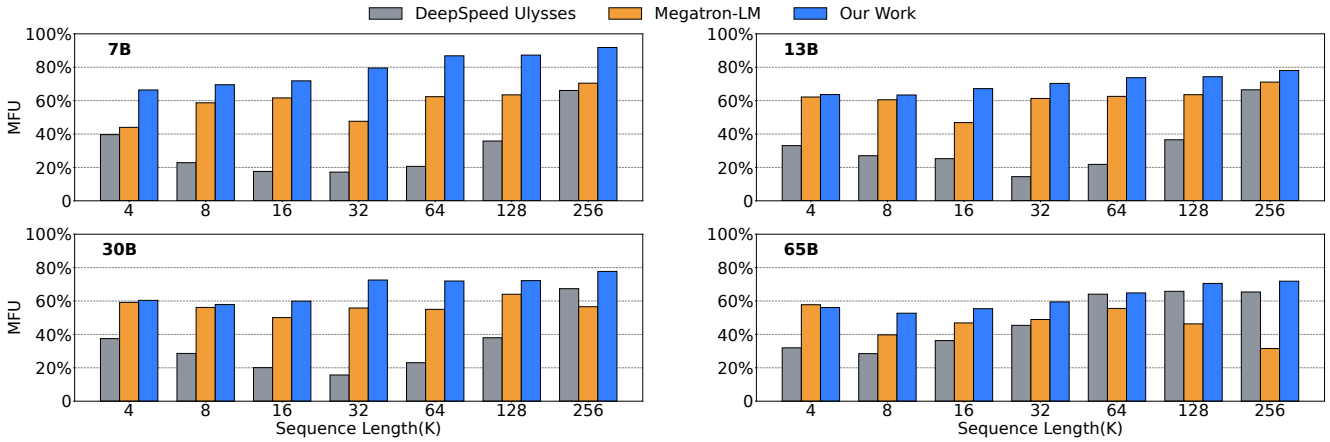

Figure 8: End-to-end evaluation results (MFU) of training different sizes of models with different sequence lengths.

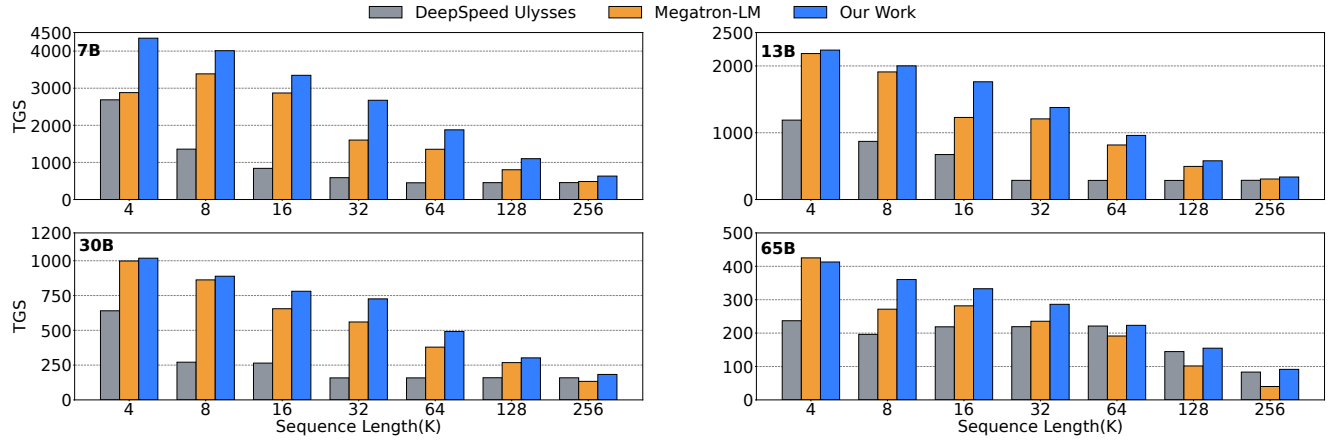

Figure 9: End-to-end evaluation results (TGS) of training different sizes of models with different sequence lengths.

decrease as the sequence length grows. By contrast, when the sequence is long enough, the computation time becomes the bottleneck. DeepSpeed Ulysses can benefit from the efficient `all-to-all` communication. The longer the sequence length ($S \geq 32K$), the more intensive the computation, and the higher the corresponding MFU.

In `InternEvo`, the communication overhead is minimized with hybrid parallelism and redundant sharding. Thus, MFU increases with the sequence length when training all four types of models. Importantly, `InternEvo` consistently outperforms baseline methods or achieves comparable training performance across all configurations. Notably, compared to DeepSpeed Ulysses and Megatron-LM, the achieved MFU by `InternEvo` stands out, surpassing them by up to 4.8× and 2.29×, respectively. This emphasizes the efficiency of `InternEvo` in managing communication overhead and optimizing training performance.

When training 30B and 65B models with a sequence length of 4k, the parallelization strategy employed by `InternEvo` coincides with that of Megatron-LM. As the sequence length in this scenario is relatively small, the volume of activations becomes much smaller than the model states. Consequently, the communication cost of Megatron-LM becomes sufficiently minimal, allowing it to achieve a high MFU. The parallelization strategy adopted by `InternEvo` in this case aligns with the optimal strategy identified by the system, resulting in an equivalent MFU for both `InternEvo` and Megatron-LM.

5.3 Ablation Study

5.3.1 Analysis of Simulator

We conduct experiments to validate the effectiveness of our simulator. Treating the set of training configurations, denoted as $\mathbb{S}$, as a ranking problem, our main concern is whether the order of these configurations, especially within the top-10 solutions, is accurately selected for real-system profiling. We select the first 50 solutions generated by the simulator and group them into top 10, top 11-20, top 21-30, top 31-40, and top 41-50. The actual TGS for these 50 configurations is measured and visualized in Figure 11.

By comparing the simulation results for 65B-4K and 65B-256K training, we find that the 65B-256K simulation provides

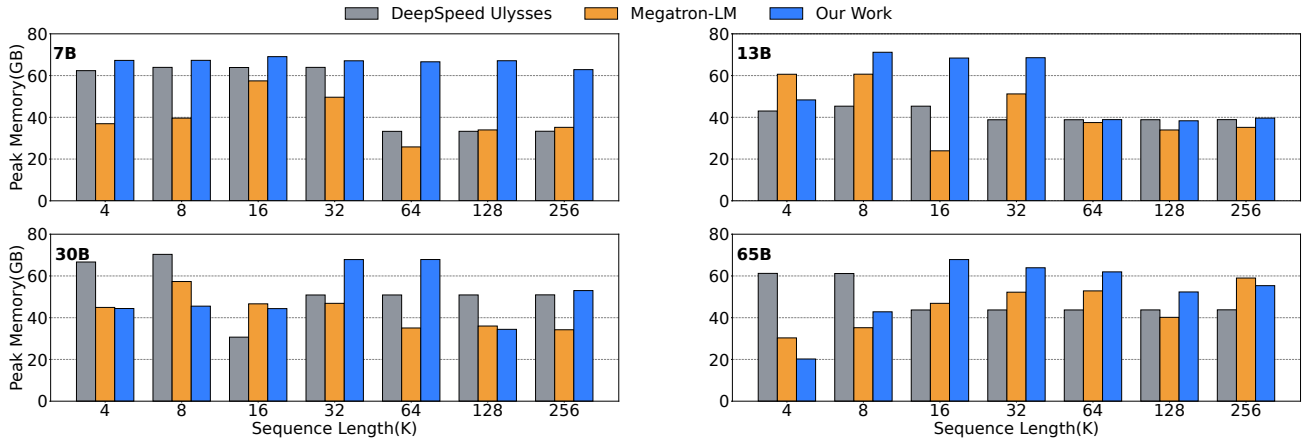

Figure 10: Peak Memory of training different sizes of models with different sequence lengths.

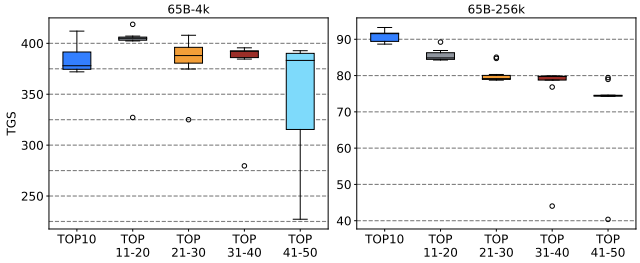

Figure 11: The performance of Top-50 configs on 65B for 4k and 256k sequence length.

a more accurate estimation, indicating the simulator's good performance with longer sequences. The observed discrepancy in the median value of top 11-20 in the 65B-4K training is due to the uncertain overlapping modeling between computation and communication, caused by frequent kernel execution and *bubbles* across kernels, especially with shorter sequences. However, this discrepancy does not impact the final deployment results. The maximum value within the top 10 configurations, representing the highest TGS among the first 50 results, is captured during deployment as all top 10 configurations are executed.

5.3.2 Analysis of Communication Scheduler

We conduct a comparison of the TGS achieved by `InternEvo` with and without the selective overlap under the same parallelization strategy. The impact of selective overlap on training the 13B model is illustrated in Figure 12(a).

The results demonstrate that the selective overlap approach leads to improvements in TGS ranging from 0.9% to 75.2% across different sequence lengths. The improvement is particularly substantial when transitioning from a sequence length of 4k to 64k. In these instances, the overall training performance is notably influenced by the communication costs without overlapping. A specific scenario exemplifying the impact of selective overlap occurs with a sequence length of 8k, where the absence of this approach results in an OOM error. This

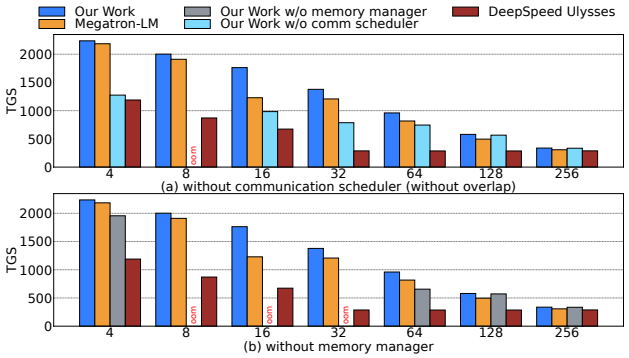

Figure 12: Effects of the Memory Management and Communication Scheduler (13B model). The x-axis indicates the sequence length (k).

occurrence can be attributed to the system's inability to leverage the specialized memory pool dedicated to communication without selective overlap.

5.3.3 Analysis of Memory Management

In order to demonstrate the efficacy of memory management techniques, including *Pinned Communication Buffer* and *Memory Consolidation in Advance*, we conduct a comparative analysis of the TGS with and without the implementation of these techniques. The results are illustrated in Figure 12(b). Without memory management, OOM errors occur when the sequence length is 8k, 16k, and 32k due to an excessive number of memory fragments. For other sequence lengths, there are no OOM errors observed, but the memory swapping (calling `CudaMalloc` and `CudaFree` due to high memory usage pressure and fragmentation) can limit the TGS.

5.4 Sequence Length Scalability

We conduct an evaluation of the largest sequence length for all three systems, configuring both b and n to 1 to facilitate the handling of longer sequences. To ensure compatibility with sequence parallelism (s_{sp}), it is crucial that the number of attention heads is divisible by s_{sp}. Additionally, we set

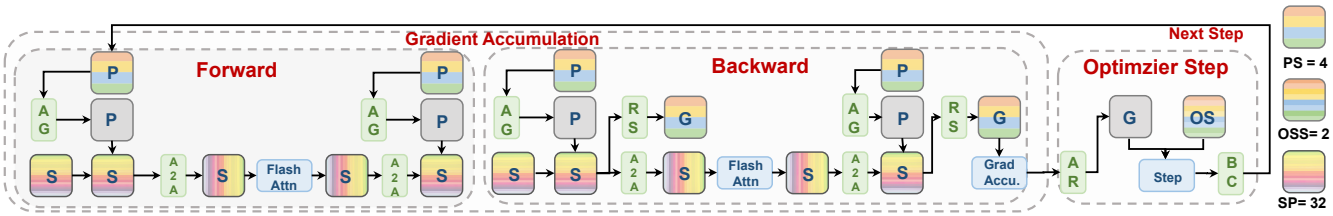

Figure 13: Visualization of parallelization strategies for training a 7B model with 256k sequence length on 128 GPUs. Different colors represent the GPUs a tensor is distributed to. Grey blocks represent a tensor that is replicated across GPUs. AG, AR, A2A, RS, and BC represent `all-gather`, `all-reduce`, `all-to-all`, `reduce-scatter`, and `broadcast` respectively.

the constraint $s_{dp} \cdot s_{sp} \cdot s_{pp} = N$ to guarantee the feasibility of s_{sp}. For the 7B and 65B models on 128 GPUs, we set s_{sp} to 32 and 64, respectively, representing their maximum s_{sp} values that support longer sequences while adhering to the attention head constraint. Furthermore, we evaluate the 13B and 30B models on 80 GPUs and 96 GPUs, setting s_{sp} to 40 and 48, respectively. These choices are made to ensure that the number of attention heads is divisible by s_{sp}.

We acknowledge that our `InternEvo` is subject to limitations imposed by the number of attention heads. However, this constraint exerts comparatively less influence on our methodologies, which support sequence lengths of up to 1536K. This is in contrast to certain existing SP-3 approaches, such as those highlighted in [30], which primarily focus on scaling beyond the number of heads.

Table 3 provides an overview of the maximum sequence lengths supported by `InternEvo` and the corresponding MFU in comparison to the baseline. `InternEvo` accommodates the same sequence lengths with the baselines while maintaining a consistent 75% MFU on average. Megatron-LM achieves an equivalent sequence length with `InternEvo` for all models. This uniformity arises due to the comparable reduction in the activation memory usage by both approaches, achieved by a factor of s_{sp}. However, Megatron-LM incurs more network communication for activations compared to `InternEvo`, resulting in slower execution. The time-out error in DeepSpeed Ulysses occurs because, while it transmits less activation data than Megatron-LM, it also communicates model states across the world size simultaneously. Without refined communication scheduling, this frequent and resource-intensive communication triggers the time-out error in DeepSpeed Ulysses.

5.5 Case Study

We present a visualization of the execution plan $\mathbb{S}$ obtained by `InternEvo` for training a 7B model with 256k sequence length on 128 GPUs in Figure 13. `InternEvo` opts not to utilize the pipeline parallelism and activation recomputation mechanism during the training. `InternEvo` sets s_{sp} as 32 to accommodate the 256k sequence on the device. It also sets s_{oss} as 2 to reduce the memory usage of the optimizer state (Adam) from 42 GB to 5 GB. Given that model states of parameters and gradients constitute a small portion of overall memory usage in 7B model, `InternEvo` further limits the communication costs by setting s_{ps} to 4, thereby restricting the number of communication participants.

Table 3: The largest sequence length supported by `InternEvo`, DeepSpeed Ulysses, and Megatron-LM (Max S). ✗means time-out error.

	DeepSpeed-U		Megatron-LM		`InternEvo`	
	Max S	MFU	Max S	MFU	Max S	MFU
7B	1536K	✗	1536K	35%	1536K	80%
13B	1200K	✗	1200K	27%	1200K	77%
30B	960K	✗	960K	19%	960K	75%
65B	512K	✗	512K	14%	512K	70%

6 Related Work

Systems for long-sequence training. Since the memory and calculation of a single device are limited, ColossalAI-SP [31] first proposes the segmentation and parallelism for the sequence dimension in addition to tensor parallelism for hidden dimension and pipeline parallelism for model depth. On this basis, Ring Attention [29, 50] uses blockwise self-attention to split long sequences into different devices and overlap the communication of key-value blocks. Recently, LightSeq [30] further improved the efficiency of long sequence modeling through load balancing for causal language modelings and a re-materialization-aware checkpointing strategy. The advantage of the above sequence parallelism is that it can break through architectural limitations and achieve infinitely long sequence modeling. In contrast, another type of sequence parallelism emphasizes maintaining the combination with existing efficient self-attention mechanisms such as FlashAttention [33, 34] to achieve input with almost infinity. For example, Megatron-LM [27] only uses sequence parallelism during Dropout and Layernorm operations, thereby reducing activation redundancy. In addition, DeepSpeed-Ulysses [28] uses an alternative all-to-all collective communication gathering weight for attention computation when segmenting the sequence, avoiding communication overhead that originally increases with length.

Systems for model-parallelism training. Parallelisms are used for distributed DNN training systems. TorchDDP [51], Horovod [52] support data parallelism. ByteScheduler [53] and DeepSpeed [54] extend data parallelism with communication and memory optimization. Tofu [55], Flexflow [23], and GSPMD [49] leverage tensor parallelism to distribute model weight across devices. Dapple [56] and Megatron-LM [22] also leverage pipeline parallelism to divide the model into multiple stages and distribute them across devices.

Systems for model-sharding training. ZeRO [39, 57], cross-replica sharding [58] has proposed to address the memory limitation issue of traditional data parallelism strategy by sharding the model states onto all GPUs. MiCS [42] minimizes the communication scale on top of ZeRO for better scalability on the public cloud by partitioning model states within subgroups. FSDP's hybrid sharding [8] implemented MiCS to Pytorch [59]. ZeRO++ [41] takes a different approach by redundantly storing an additional set of secondary parameters on each node, in exchange for enhanced communication efficiency through parameter pre-fetching. AMSP [60] takes a more flexible approach by partitioning the three components of model states independently for better scalability in the training of not-so-large (7B and 13B) models on thousands of GPUs. PaRO [61] provides a fine-grained sharding strategy for various training scenarios like parameter-efficient fine-tuning task [62, 63].

Automatic search for distributed training. For general models and hardware, some systems [9, 64, 65] employ mathematical programming techniques, such as dynamic programming and integer linear programming, to automate the configuration of parallel mechanisms. To make the immense search space more manageable, these systems simplify the problem in various ways. Flexflow [23] focuses on specific mechanisms like data and tensor parallelism, Gpipe [6] and PipeDream [21] concentrates ZeRO on data and pipeline parallelism, and AMSP [60] only focus on data parallelism, Alpa [9] and Galvatron [24] focus on all parallelism except sequence parallelism.

7 Conclusion

In this paper, we propose `InternEvo`, an automatic parallelization framework for training LLMs with long sequences. We first decouple the GPU memory management dimensions of LLM training into a brand new hierarchical space and systematically analyze their memory and communication costs. Then we develop an execution simulator to derive an efficient parallelization strategy in this hierarchical space. We design a selective overlap method and a memory fragmentation manager to further improve the computation resource utilization and memory utilization. Evaluation results show that `InternEvo` can generate the optimal parallelization strategies that match or outperform existing methods in MFU.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of 26th Annual Conference on Neural Information Processing Systems, NeurIPS 2012.*, pages 1106–1114, 2012.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[3] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014*, pages 583–598, 2014.

[4] Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Proceedings of 28th Annual Conference on Neural Information Processing Systems, NeurIPS 2014.*, pages 19–27, 2014.

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of 26th Annual Conference on Neural Information Processing Systems, NeurIPS 2012.*, pages 1232–1240, 2012.

[6] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, NIPS'19. Curran Associates Inc., 2019.

[7] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of 17th European Conference on Computer Systems, EuroSys 2022*, pages 472–487, 2022.

[8] 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, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *CoRR*, abs/2304.11277, 2023.

[9] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *CoRR*, abs/2201.12023, 2022.

[10] Jinjie Ni, Tom Young, Vlad Pandelea, Fuzhao Xue, and Erik Cambria. Recent advances in deep learning based dialogue systems: A systematic survey. *Artificial intelligence review*, 56(4):3055–3155, 2023.

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

[12] Wenxuan Zhou, Kevin Huang, Tengyu Ma, and Jing Huang. Document-level relation extraction with adaptive thresholding and localized context pooling. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 14612–14620, 2021.

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

[14] Hao Zhang, Aixin Sun, Wei Jing, and Joey Tianyi Zhou. Span-based localizing network for natural language video localization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6543–6554, 2020.

[15] Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lučić, and Cordelia Schmid. Vivit: A video vision transformer. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6836–6846, 2021.

[16] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zi-Hang Jiang, Francis EH Tay, Jiashi Feng, and Shuicheng Yan. Tokens-to-token vit: Training vision transformers from scratch on imagenet. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 558–567, 2021.

[17] Kaifeng Bi, Lingxi Xie, Hengheng Zhang, Xin Chen, Xiaotao Gu, and Qi Tian. Accurate medium-range global weather forecasting with 3d neural networks. *Nature*, 619(7970):533–538, 2023.

[18] Microsoft Azure Quantum Microsoft Research AI4Science. The impact of large language models on scientific discovery: a preliminary study using gpt-4. *arXiv preprint arXiv:2311.07361*, 2023.

[19] Ludan Ruan and Qin Jin. Survey: Transformer based video-language pre-training. *AI Open*, 3:1–13, 2022.

[20] Abel Chandra, Laura Tünnermann, Tommy Löfstedt, and Regina Gratz. Transformer-based deep learning for predicting protein properties in the life sciences. *Elife*, 12:e82819, 2023.

[21] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19. Association for Computing Machinery, 2019.

[22] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. *CoRR*, abs/2104.04473, 2021.

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018.

[24] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proceedings of the VLDB Endowment*, 16:470–479, 2022.

[25] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1466–1478, 2023.

[26] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 267–284. USENIX Association, 2022.

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

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

[29] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *CoRR*, abs/2310.01889, 2023.

[30] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Joseph E Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. Lightseq: Sequence level parallelism for distributed training of long context transformers. *arXiv preprint arXiv:2310.03294*, 2023.

[31] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. *CoRR*, abs/2105.13120, 2022.

[32] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *CoRR*, abs/2205.14135, 2022.

[33] PyTorch. Accelerating large language models with accelerated transformers. https://pytorch.org/blog/accelerating-large-language-models/, 2023.

[34] Patrick von Platen. Optimizing your llm in production. https://huggingface.co/blog/optimize-llm, 2023.

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[36] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20. Curran Associates Inc., 2020.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017.

[38] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2017.

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

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

[41] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Zero++: Extremely efficient collective communication for giant model training. *CoRR*, abs/2306.10209, 2023.

[42] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. Mics: near-linear scaling for training gigantic model on public cloud. *Proceedings of the VLDB Endowment*, 16:37–50, 2022.

[43] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, Haoran Zhang, and Jie Zhao. Oneflow: Redesign the distributed deep learning framework from scratch. *CoRR*, abs/2110.15032, 2022.

[44] Yonghao Zhuang, Hexu Zhao, Lianmin Zheng, Zhuohan Li, Eric Xing, Qirong Ho, Joseph Gonzalez, Ion Stoica, and Hao Zhang. On optimizing the communication of model parallelism. *Proceedings of Machine Learning and Systems*, 5, 2023.

[45] NVIDIA Developers. Nccl user guide: Allreduce. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/operations.html#allreduce, 2023.

[46] NVIDIA Developers. Fast, flexible allocation for nvidia cuda with rapids memory manage. https://developer.nvidia.com/blog/fast-flexible-allocation-for-cuda-with-rapids-memory-manager/, 2020.

[47] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling

with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[48] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.

[49] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020.

[50] Hao Liu and Pieter Abbeel. Blockwise parallel transformer for large context models. *CoRR*, abs/2305.19370, 2023.

[51] torch. Torchddp. https://pytorch.org/, 2023.

[52] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[53] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20. Association for Computing Machinery, 2020.

[55] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. Association for Computing Machinery, 2019.

[56] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21. Association for Computing Machinery, 2021.

[57] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *CoRR*, abs/2104.07857, 2021.

[58] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training. *CoRR*, abs/2004.13336, 2020.

[59] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *CoRR*, abs/2006.15704, 2020.

[60] Qiaoling Chen, Qinghao Hu, Zhisheng Ye, Guoteng Wang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Amsp: Super-scaling llm training via advanced model states partitioning. *CoRR*, abs/2311.00257, 2023.

[61] Chan Wu, Hanxiao Zhang, Lin Ju, Jinjing Huang, Youshao Xiao, Zhaoxin Huan, Siyuan Li, Fanzhuang Meng, Lei Liang, Xiaolu Zhang, and Jun Zhou. Rethinking memory and communication cost for efficient large language model training. *CoRR*, abs/2310.06003, 2023.

[62] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685, 2021.

[63] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *CoRR*, abs/2110.07602, 2022.

[64] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous GPUs. In *2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 673–688. USENIX Association, 2022.

[65] Jakub M. Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In *Advances in Neural Information Processing Systems*, NeurIPS '21, 2021-12-06.

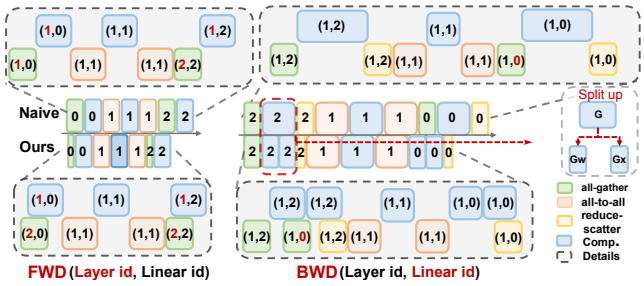

Figure 14: Comparison of `InternEvo`'s overlap with a naive pass in two layers, one attention, and three linear calculations in each layer. We use different overlap granularities in the forward and backward passes.

A Appendix

A.1 Implementation Detail: Selective Overlap

The Communication Scheduler module within `InternEvo` introduces an innovative method designed to maximize the utilization of idle computation resources by strategically overlapping communication and computation processes. This overlap can manifest at either the inter-layer or intra-layer granularity. Although the intra-layer overlap strategy initially seems appealing due to its finer granularity and lower memory requirements for communication buffers, a closer examination reveals potential drawbacks.

A straightforward yet limited approach might involve consistently applying the intra-layer overlap strategy. However, this fine-grained strategy, while reducing memory demands, involves more frequent communication instances. In scenarios with CPU-intensive workloads, the CPU thread may struggle to timely issue every communication operator. Consequently, the cumulative latency introduced by CPU scheduling and communication may surpass the execution time of computation. This limitation indicates that intra-layer overlap alone may not consistently mask communication overhead effectively. Therefore, a judicious consideration of workload characteristics becomes crucial, necessitating an adaptive approach to choosing between inter-layer and intra-layer overlap strategies for optimizing overall system performance.

Derived from this solution is the recognition that the key to achieving our goal is to decide the overlap granularity based on actual communication latency and computation execution time. Notably, the observation that backward propagation involves computation twice as intensive as forward propagation inspires the adoption of a nuanced strategy called **selective overlap**. In this approach, a memory-efficient, fine-grained overlap is employed during backward passes, capitalizing on the longer computation duration that can tolerate greater latency. Conversely, for the forward pass, the inter-layer overlap strategy is adopted, trading off increased memory usage for a significantly smaller step time. This iterative execution of inter-layer and intra-layer communication-computation overlap aims to strike a balance between computational efficiency and memory utilization, contributing to an overall enhancement in system performance.

Intra-Layer Overlap in Backward. In the backward pass, we distinguish between two computation categories: *G-X* (gradient with respect to the input) and *G-W* (gradient with respect to the layer's weight), as depicted in the top right corner of Figure 14. While the conventional approach combines *G-X* and *G-W* into a single backward function for user-friendliness, it inadvertently introduces inefficiencies. Specifically, the `reduce-scatter` communication-related to *G-W* is independent of *G-X*, leading to unnecessary delays in the critical path. To address this, we propose an intra-layer overlap strategy. This allows *G-X* computations to overlap with `reduce-scatter` communication for *G-W*. Additionally, we introduce backward pre-fetching, launching the next `all-gather` before the current `reduce-scatter` in the layer. This asynchronous, non-blocking task runs concurrently with *G-W* computation. An important insight guiding this strategy is that the communication order in the forward pass is the reverse of the backward pass. Leveraging this, we can anticipate which parameters to `all-gather` next in backward pre-fetching. This orchestrated approach enhances communication-computation overlap, improving overall performance during backward propagation.

Inter-layer overlap in forward. Utilizing an overlap policy with module (such as linear module) granularity could result in delays for the next `all-gather` kernel to start, subsequently blocking computations on critical paths that depend on it. Such blocking effects tend to accumulate within a layer. To address this, we adopt an inter-layer overlap strategy, aiming to mitigate the impact of kernel launch latency on forward propagation. This strategy involves obtaining parameters for the next layer in advance while computing the current layer. As depicted in Figure 14 in the forward pass, we implement a forward pre-fetch for subsequent layers. Specifically, we launch the `all-gather` for the second layer $(2,0)$ and schedule this collective communication to run concurrently with the corresponding linear module computation task of the first layer. This inter-layer overlap approach optimizes the forward pass by proactively fetching parameters for upcoming layers, reducing latency and enhancing overall efficiency.

A.2 Implementation Detail: Defragmentation

In this section, we discuss the design of *Pinned Communication Buffer* and *Memory Consolidation in Advance* for reducing GPU memory fragmentation.

Pinned Communication Buffer. In long-sequence training, the continuous memory demand for activation generated by the computation stream increases. However, frequent `all-gather` communications for parameter pre-fetching may lead to GPU memory fragmentation. To tackle this, `InternEvo` introduces memory pools dedicated to `all-gather` in forward and backward passes. Before forward propagation, `InternEvo` allocates a new pool with a

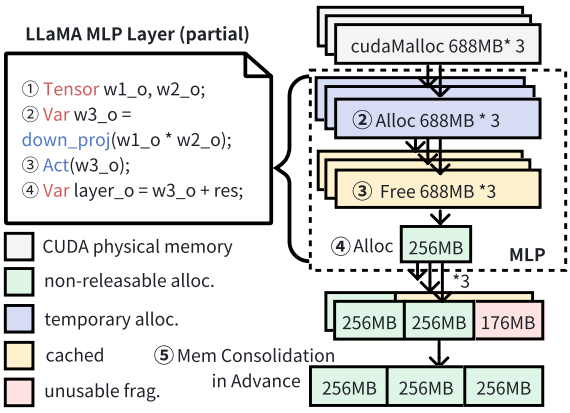

Figure 15: We illustrate the necessity of defragmentation techniques using the 65B 256K configuration from Table 7 as an example. Owing to the gate projection, up projection (①), and the dot product of these projections (②), there are three allocation requests of 688MiB each, with a shape of (16384, 22016) within one MLP layer. Next, the activation checkpoint mechanism releases these intermediate activations (③); however, the MLP output is maintained as a non-releasable tensor until the corresponding backward stage (④). As the PyTorch memory allocator employs the BestFit strategy for tensor allocation [2] and the segments of 688MiB have been cached, the 256MiB MLP output is likely placed into those 688MiB segments, resulting in a fragmentation of 176MiB per segment. Such fragmentation is too small to reuse in the 65B model forward stage with a 256K sequence. In the worst-case scenario, these unrecyclable fragments could accumulate to as much as 5.32 GiB. To tackle this significant memory challenge, we proactively aggregate and flatten outputs from every three MLP layers to avert subsequent OOM (⑤). Our experiments demonstrate that this anti-fragmentation approach effectively reduces fragmentation at an acceptable cost, thereby ensuring trainability.

size equal to the required size for two layers' `all-gather` communication buffer. It implements double buffer rotation, utilizing one buffer for computation and another for communication in an overlapping mode. Therefore, `InternEvo` would not request memory for `all-gather` from PyTorch on the demand during training. This approach helps manage memory efficiently and addresses fragmentation concerns caused by frequent communication operations.

Memory Consolidation in Advance. To prevent potential out-of-memory (OOM) issues caused by fragmentation, we adopt a strategy of consolidating small, scattered memory chunks through heuristic methods. Transformer models, especially when using the standard LLaMa model with `MLP_Ratio 8/3` and activation recomputation, tend to experience increased internal fragmentation in MLP layers due to irregular dimensions. Our solution involves combining these irregular and small output segments to reduce fragmentation, thus preventing OOM incidents. Figure 15 gives an example of such a case. Additionally, we've recognized challenges posed by fragmented memory space, particularly occupied by gradients, especially in long sequence training. To tackle this, prior to each forward operation at every step, we proactively map the `.grad` attribute of each learnable parameter to a contiguous memory chunk.

A.3 Solution Algorithm

Algorithm 1 encapsulates the workflow employed by `InternEvo` to derive optimal execution plan candidates. The execution plan $\mathbb{S} = [b, n, a, s_{pp}, s_{dp}, s_{tp}, s_{sp}, s_{ps}, s_{gs}, s_{oss}]$ is systematically explored, with `InternEvo` considering all possible combinations of micro-batch size, micro-batch number, parallelism strategies, and sharding strategies. For a given candidate solution $\mathbb{S}_t$, `InternEvo` initiates the process by estimating its memory usage and rigorously checking its adherence to constraints such as single GPU memory capacity, global batch size, and global GPU number. If $\mathbb{S}_t$ proves feasible, `InternEvo` proceeds to estimate its execution time utilizing existing profiling data and default device mesh placement sequences. The algorithm dynamically maintains a curated list of the top 10 solutions, continuously updating it throughout the search iterations based on the lowest observed execution times. The final output comprises the top 10 optimal execution plans.

Algorithm 1 Search Optimal Execution Plan

Input: Model Architecture, GPU Count N
Output: Top10 Execution Plans
$\mathbb{S} = [b, n, a, s_{pp}, s_{dp}, s_{tp}, s_{sp}, s_{ps}, s_{gs}, s_{oss}]$
Initialization: top10_solutions = None
Initialization: top-10_cost = ∞
for $s_{pp} \in \{1, 2, \ldots, N\}$ **do**
  for $s_{sp}, s_{tp} \in \{1, 2, 4, \ldots, N/s_{pp}\}$ **do**
    for s_{dp} **in** $\{1, 2, \ldots, N/s_{pp} \cdot s_{tp}\}$ **do**
      for b **in** $\{1, 2, \ldots, B/(S \cdot s_{dp})\}$ **do**
        for s_{ps},s_{gs},s_{oss} **in** $\{1, 2, \ldots, N\}$ **do**
          Get micro-batch num $n = B/(S \cdot s_{dp} \cdot b)$
          Create candidate solution $\mathbb{S}_t$
          if CHECKCONSTRAINTS($\mathbb{S}_t$) **then**
            Estimate $T_{fwd\_bwd}(\mathbb{S}_t) + T_{update}(\mathbb{S}_t)$
            if $T(\mathbb{S}_t) <$ max(top10_time) **then**
              Update top10_solutions
Print Best Solution: best_solutions

A.4 Details of E2E Evaluation

Table 4 to 7 shows the detailed parallel configs in Section 5. Figure 10 shows the peak memory of different sizes of models with different sequence lengths.

Table 4: Configuration on Model 7B

Seq	System	n	b	s_{pp}	s_{dp}	s_{tp}	s_{sp}	s_{ps}	s_{oss}	a
4k	Megatron-LM	8	2	1	64	2	2	1	64	0
	Deepspeed-U	4	2	1	128	1	1	128	1	0
	Our Work	4	2	1	128	1	1	4	2	0
8k	Megatron-LM	8	1	1	64	2	2	1	8	0
	Deepspeed-U	16	2	1	64	1	2	128	1	0
	Our Work	4	1	1	128	1	1	4	2	0
16k	Megatron-LM	4	1	1	64	2	2	1	64	0
	Deepspeed-U	4	4	1	16	1	8	128	1	0
	Our Work	4	1	1	64	1	2	2	4	0
32k	Megatron-LM	4	1	1	32	4	4	1	32	0
	Deepspeed-U	4	2	1	16	1	8	128	1	0
	Our Work	4	1	1	32	1	4	2	4	0
64k	Megatron	1	2	1	32	4	4	1	32	1
	Deepspeed-U	1	4	1	16	1	8	128	1	1
	Our Work	4	1	1	16	1	8	2	4	0
128k	Megatron-LM	1	2	1	16	8	8	1	1	1
	Deepspeed-U	2	2	1	16	1	8	128	1	1
	Our Work	4	1	1	8	1	16	2	4	0
256k	Megatron-LM	4	1	1	16	8	8	1	4	1
	Deepspeed-U	1	1	1	16	1	8	128	1	1
	Our Work	4	1	1	4	1	32	4	2	0

Table 5: Configuration on Model 13B

Seq	System	n	b	s_{pp}	s_{dp}	s_{tp}	s_{sp}	s_{ps}	s_{oss}	a
4k	Megatron-LM	8	2	1	64	2	2	1	64	0
	Deepspeed-U	8	1	1	128	1	1	128	1	0
	Our Work	16	1	2	64	1	1	1	16	0
8k	Megatron-LM	8	1	1	64	2	2	1	64	0
	Deepspeed-U	8	2	1	32	1	4	128	1	0
	Our Work	4	8	1	16	1	8	128	1	0
16k	Megatron-LM	16	1	1	16	8	8	1	16	0
	Deepspeed-U	8	2	1	16	1	8	128	1	0
	Our Work	8	1	1	32	1	4	4	8	0
32k	Megatron-LM	1	2	1	64	2	2	1	64	1
	Deepspeed-U	1	8	1	16	1	8	128	1	1
	Our Work	8	1	1	16	1	8	2	8	0
64k	Megatron-LM	1	2	1	32	4	4	1	32	1
	Deepspeed-U	1	4	1	32	1	4	128	1	1
	Our Work	8	1	2	8	1	8	64	1	0
128k	Megatron-LM	1	2	1	16	8	8	1	16	1
	Deepspeed-U	1	2	1	16	1	8	128	1	1
	Our Work	1	1	1	32	1	4	16	1	1
256k	Megatron-LM	1	1	1	16	8	8	1	16	1
	Deepspeed-U	1	1	1	16	1	8	128	1	1
	Our Work	1	1	1	16	1	8	16	1	1

Table 6: Configuration on Model 30B

Seq	System	n	b	s_{pp}	s_{dp}	s_{tp}	s_{sp}	s_{ps}	s_{oss}	a
4k	Megatron-LM	32	1	2	32	2	2	1	32	0
	Deepspeed-U	8	1	1	128	1	1	128	1	0
	Our Work	32	1	2	32	2	2	1	32	0
8k	Megatron-LM	16	1	1	32	4	4	1	32	0
	Deepspeed-U	8	4	1	16	1	8	128	1	0
	Our Work	64	1	4	8	1	4	1	8	0
16k	Megatron-LM	2	4	1	32	4	4	1	32	1
	Deepspeed-U	2	8	1	16	1	8	128	1	1
	Our Work	64	1	4	4	1	8	1	8	0
32k	Megatron-LM	2	2	1	32	4	4	1	32	1
	Deepspeed-U	1	8	1	16	1	8	128	1	1
	Our Work	8	1	1	16	1	8	32	4	0
64k	Megatron-LM	2	2	1	16	8	8	1	16	1
	Deepspeed-U	1	4	1	16	1	8	128	1	1
	Our Work	8	1	1	8	1	16	32	4	0
128k	Megatron-LM	2	1	1	16	8	8	1	16	1
	Deepspeed-U	1	2	1	16	1	8	128	1	1
	Our Work	2	1	1	16	1	8	32	1	1
256k	Megatron-LM	2	1	1	8	16	16	1	8	1
	Deepspeed-U	1	1	1	16	1	8	128	1	1
	Our Work	1	1	1	16	1	8	32	1	1

Table 7: Configuration on Model 65B

Seq	System	n	b	s_{pp}	s_{dp}	s_{tp}	s_{sp}	s_{ps}	s_{oss}	a
4k	Megatron-LM	128	1	8	8	2	2	1	8	0
	Deepspeed-U	16	1	1	64	1	2	128	1	0
	Our Work	256	1	16	4	2	2	1	4	0
8k	Megatron-LM	64	1	4	8	4	4	1	8	0
	Deepspeed-U	16	2	1	16	1	8	128	1	0
	Our Work	256	1	8	2	1	8	1	16	0
16k	Megatron-LM	8	2	2	16	4	4	1	16	1
	Deepspeed-U	4	4	1	32	1	4	128	1	1
	Our Work	4	1	1	64	1	2	8	8	1
32k	Megatron-LM	4	2	1	16	8	8	1	16	1
	Deepspeed-U	2	2	1	32	1	4	128	1	1
	Our Work	4	1	1	32	1	4	8	8	1
64k	Megatron-LM	4	1	1	16	8	8	1	16	1
	Deepspeed-U	2	1	1	32	1	4	128	1	1
	Our Work	4	1	1	16	1	8	8	8	1
128k	Megatron-LM	4	1	1	8	16	16	1	8	1
	Deepspeed-U	2	1	1	16	1	8	128	1	1
	Our Work	2	1	1	16	1	8	32	2	1
256k	Megatron-LM	2	1	1	8	16	16	1	8	1
	Deepspeed-U	2	1	1	8	1	16	128	1	1
	Our Work	2	1	1	8	1	16	32	2	1